

AD-A255 978

0

Wait-Free Consensus

James Aspnes

July 24, 1992

CMU-CS-92-164

DTIC  
ELECTE  
OCT 07 1992  
S A D

School of Computer Science  
Carnegie Mellon University  
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements  
for the degree of Doctor of Philosophy

This document has been approved  
for public release and sale; its  
distribution is unlimited.

©1992 James Aspnes.

This research was supported by an IBM Graduate Fellowship and an NSF Graduate Fellowship.

92 10 5 028

423887 92-26537 19  
19 pg 7

**Keywords:** distributed algorithms, shared memory, consensus, random walks, martingales, shared coins



School of Computer Science

DOCTORAL THESIS  
in the field of  
Computer Science

*Wait-Free Consensus*

JAMES ASPNES

Submitted in Partial Fulfillment of the Requirements  
for the Degree of Doctor of Philosophy

ACCEPTED:

A handwritten signature of a person, likely the Major Professor.

MAJOR PROFESSOR

7/21/92

DATE

A handwritten signature of a person, likely the Dean.

DEAN

8/24/92

DATE

APPROVED:

A handwritten signature of a person, likely the Provost.

PROVOST

25 August 1992

DATE

Accession For	
NTIS	CRA&I <input checked="" type="checkbox"/>
DTIC	TAB <input type="checkbox"/>
Unannounced <input type="checkbox"/>	
Justification	
By <i>per ltr.</i>	
Distribution /	
Availability	
Dist	Avail <i>Exhibit</i>
A-1	

DTIC QUALITY INSPECTED 1

### Abstract

Consensus is a decision problem in which  $n$  processors, each starting with a value not known to the others, must collectively agree on a single value. If the initial values are equal, the processors must agree on that common value; this is the **validity** condition. A consensus protocol is **wait-free** if every processor finishes in a finite number of its own steps regardless of the relative speeds of the other processors, a condition that precludes the use of traditional synchronization techniques such as critical sections, locking, or leader election. Wait-free consensus is fundamental to synchronization without mutual exclusion, as it can be used to construct wait-free implementations of arbitrary concurrent data structures. It is known that no deterministic algorithm for wait-free consensus is possible, although many randomized algorithms have been proposed.

I present two algorithms for solving the wait-free consensus problem in the standard asynchronous shared-memory model. The first is a very simple protocol based on a random walk. The second is a protocol based on weighted voting, in which each processor executes  $O(n \log^2 n)$  expected operations. This bound is close to the trivial lower bound of  $\Omega(n)$ , and it substantially improves on the best previously-known bound of  $O(n^2 \log n)$ , due to Bracha and Rachman.

# Acknowledgments

I would like to begin by thanking the members of my committee. My advisor, Steven Rudich, has always been willing to provide encouragement. Danny Sleator showed me that research is better done as play than as work. Merrick Furst has always been a source of interesting observations and ideas. Maurice Herlihy introduced me to wait-free consensus when I first arrived at Carnegie Mellon; his keen insights into distributed computing have been a continuing influence on my work.

I would like to thank my parents for their warm support and encouragement.

I would like to thank the many other students who lightened the monastic burdens of graduate student life. David Applegate in particular was always ready to supply a new problem or a new toy.

Finally, I would like to thank my beloved wife, Nan Ellman, who waited longer and more patiently for me to finish than anyone.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>The Asynchronous Shared-Memory Model</b>	<b>8</b>
2.1	Basic elements . . . . .	8
2.2	Time and asynchrony . . . . .	9
2.3	Randomization . . . . .	10
2.4	Relation to other models . . . . .	10
2.5	Performance measures . . . . .	12
<b>3</b>	<b>Consensus and Shared Coins</b>	<b>14</b>
3.1	Consensus . . . . .	14
3.2	Shared coins . . . . .	16
3.3	Consensus using shared coins . . . . .	17
<b>4</b>	<b>Consensus Using a Random Walk</b>	<b>20</b>
4.1	Random walks . . . . .	21
4.2	The robust shared coin protocol . . . . .	23
4.3	Implementing a bounded counter with atomic registers . . . . .	29
4.4	The randomized consensus protocol . . . . .	31
<b>5</b>	<b>Consensus Using Weighted Voting</b>	<b>36</b>
5.1	Introduction . . . . .	36
5.2	The shared coin protocol . . . . .	38
5.3	Martingales . . . . .	40
5.3.1	Knowledge, $\sigma$ -algebras, and measurability . . . . .	41
5.3.2	Definition of a martingale . . . . .	42
5.3.3	Gambling systems . . . . .	43

5.3.4	Limit theorems	43
5.4	Proof of correctness	46
5.4.1	Phases of the protocol	47
5.4.2	Common votes	50
5.4.3	Extra votes	52
5.4.4	Written votes vs. decided votes	55
5.4.5	Choice of parameters	56
<b>6</b>	<b>Conclusions and Open Problems</b>	<b>60</b>
6.1	Comparison with other protocols	60
6.2	Limits to wait-free consensus	62
6.3	Open problems	63

# List of Figures

3.1	Consensus from a shared coin. . . . .	18
4.1	Robust shared coin protocol. . . . .	24
4.2	Pictorial representation of robust shared coin protocol. . . . .	24
4.3	The protocol as a controlled random walk. . . . .	27
4.4	Pseudocode for counter operations. . . . .	30
4.5	Consensus protocol. . . . .	32
4.6	Counter scan for randomized consensus protocol. . . . .	32
5.1	Shared coin protocol. . . . .	38

# List of Tables

6.1 Comparison of consensus protocols. . . . .	61
--	----

# Chapter 1

## Introduction

**Consensus** [CIL87] is a tool for allowing a group of processors to collectively choose one value from a set of alternatives. It is defined as a decision problem in which  $n$  processors, each starting with a value (0 or 1) not known to the others, must collectively agree on a single value. (The restriction to a single bit does not prevent the processors from choosing between more than two possibilities since they can run just run a one-bit consensus protocol multiple times.) The processors communicate by reading from and writing to a collection of registers; each processor finishes the protocol by deciding on a value and halting. A consensus protocol is **wait-free** if each processor makes its decision after a finite number of its *own* steps, regardless of the relative speeds or halting failures of the other processors. In addition, a consensus protocol must satisfy the **validity** condition: if every processor starts with the same input value, every processor decides on that value. This condition excludes trivial protocols such as one where every processor always decides 0.

The asynchronous shared-memory model is an attempt to capture the effect of making the weakest possible assumptions about the timing of events in a distributed system. At each moment an adversary scheduler chooses one of the  $n$  processors to run. No guarantees are made about the scheduler's choices— it may start and stop processors at will, based on a total knowledge of the state of the system, including the contents of the registers, the programming of the processors, and even the internal states of the processors. Since the scheduler can always simulate a halting failure by choosing not to run a processor, the model effectively allows up to  $n - 1$  halting failures. The

adversary's power, however, is not unlimited. It cannot cause the processors to deviate from their programming or cause operations on the registers to fail or return incorrect values.

Combined with the requirement that a consensus protocol terminate after a finite number of operations, the adversary's power precludes the use of traditional synchronization techniques such as critical sections, locking, or leader election: any processor that obtains a critical resource can be killed, and as soon as any processor or group of processors is given control over the outcome of the protocol, the scheduler can put them to sleep and leave the other processors helpless to complete the protocol on their own. In general, any protocol depending on **mutual exclusion**, where one processor's possession of a resource or role depends on other processors being excluded from it, will not be wait-free.

Wait-free consensus is fundamental to synchronization without mutual exclusion and thus lies at the heart of the more general problem of constructing highly concurrent data structures [Her91]. It can be used to obtain wait-free implementations of arbitrary abstract data types with atomic operations [Her91, Plo89]. It is also **complete for distributed decision tasks** [CM89] in the sense that it can be used to solve all such decision tasks that have a wait-free solution. Intuitively, the processors can individually simulate a sequence of operations on an object or the computation of a decision value, and use consensus protocols to choose among the possibly distinct outcomes. Conversely, if consensus is *not* possible, it is also impossible to construct wait-free implementations for many simple abstract data types, including queues, test-and-set bits, or compare-and-swap registers, as there exist simple deterministic consensus protocols (for bounded numbers of processors) using these primitives [Her91].

Alas, given the powerful adversary of the asynchronous shared-memory model it is not possible to have a **deterministic** wait-free consensus protocol, one in which the behavior of the processors is predictable in advance. In fact, in a wide variety of asynchronous models it has been shown there is no deterministic consensus protocol that works against a scheduler that can stop even a single processor [CIL87, DDS87, FLP85, Her91, LAA87, TM89]. Though this result is usually proved using more general techniques, when only *single-writer* registers are used it has a simple proof that illustrates many of the problems that arise when trying to solve wait-free consensus.

Imagine that two processors *A* and *B* are trying to solve the consensus

problem. Their situation is very much like the situation of two people facing each other in a narrow hallway; neither person has any stake in whether they pass on the left or the right, but if one goes left and the other right they will bump into each other and make no progress. When  $A$  and  $B$  are deterministic processors under the control of a malicious adversary scheduler, we can show the scheduler will be able to use its knowledge of their state and its control over the timing of events to keep  $A$  and  $B$  oscillating back and forth forever between the two possible decision values.

Here is what happens. Since each processor is deterministic, at any given point in time it has some **preference**, defined as the value ("left" or "right" in the hallway example) that it will eventually choose if the other processor executes no more operations [AH90a].<sup>1</sup> At the beginning of the protocol, each processor's preference is equal to its input, because without knowing that some other processor has a different input it must cautiously decide on its own input to avoid violating the validity condition. So we can assume that initially processor  $A$  prefers to pass on the left, and processor  $B$  on the right.

Now the scheduler goes to work. It stops  $B$  and runs  $A$  by itself. After some finite number of steps,  $A$  must make a decision (to go left) and halt, or the termination condition will be violated. But before  $A$  can finish, it must make sure that  $B$  will make the same decision it makes, or the consistency condition will be violated. So at some point  $A$  must tell  $B$  something that will cause  $B$  to change its preference to "left", and in the shared-memory model this message must take the form of a write operation (since  $B$  can't see when  $A$  does a read operation). Immediately *before*  $A$  carries out this critical write, the scheduler stops  $A$  and starts  $B$ .

This action puts  $B$  in the same situation that  $A$  was in.  $B$  still prefers to go right, and after some finite number of steps it must tell  $A$  to change its preference to "right". When this point is reached either one of two conditions holds: either  $B$  has done something to neutralize  $A$ 's still undelivered demand that  $B$  change its preference, in which case the scheduler just stops  $B$  and runs  $A$  again, or both  $A$  and  $B$  are about to deliver writes that will cause the other to change its preference. In this case, the scheduler allows both of

---

<sup>1</sup>This unfortunate possibility is unlikely to occur in the real-world hallway situation, assuming healthy participants, but it is allowed by the asynchronous shared-memory model since the adversary can always choose never to run the other processor again.

the writes to go through, and now  $A$  prefers to go right and  $B$  prefers to go left, putting the two processors back where they started with roles reversed. In effect, the adversary uses its power over the timing of events to make sure that just when  $A$  gives in and agrees to adopt  $B$ 's position,  $B$  does exactly the same thing, and so on *ad infinitum*.

Fortunately, human beings do not appear to be controlled by an adversary scheduler, so in real life one hardly ever sees two people bouncing in unison from one side of a hallway to the other for more than a few iterations. Processors that do not have even the illusion of free will can nonetheless get some of the same effect using randomization. Imagine in the hallway situation that  $A$  had the ability to tell  $B$  to flip a fair coin to set its new preference. No matter what  $A$ 's preference was (or changed to), there would be a 50% chance that the result of  $B$ 's coin flip would match  $A$ 's preference. In fact, if both processors were continually flipping coins to change their preferred value, a run of identical coin flips would soon occur that was long enough that the two processors would be able to notice that they were in agreement, and the protocol would terminate. Though this rough description leaves out many important details, it gives the basic idea behind the first randomized consensus protocol for the asynchronous shared-memory model, due to Abrahamson [Abr88]. The only drawback of the approach is that it does not scale well; the odds of  $n$  processors simultaneously flipping heads is exponentially small, and because agreement is not detected immediately in Abrahamson's protocol, its worst-case expected running time is only bounded by  $2^{O(n^2)}$ .

The first polynomial-time consensus protocol for this model is described by Aspnes and Herlihy [AH90a]. The key observation, similar to one made by Chor, Merritt, and Shmoys [CMS89] in the context of a different model, is that the  $n$  different local coin flips can be replaced by a single **shared coin protocol** that produces random bits that all of the processors agree on with at least a constant probability, regardless of the behavior of the scheduler. We showed that it is possible to construct a consensus protocol from any shared coin protocol by running the shared coin repeatedly until agreement is reached. (A description of the construction appears in Section 3.3.) The cost of the resulting consensus protocol is within a constant factor of the cost of the shared coin. Subsequent work on shared-memory consensus protocols has concentrated primarily on the problem of constructing efficient shared coin protocols.

All currently known shared coin protocols use some form of a very simple

idea. Each processor repeatedly adds random  $\pm 1$  votes to a common pool until some termination condition is reached. Any processor that sees a positive total vote decides 1, and those that see a negative total vote decide 0. Intuitively, because all of the processors are executing the same loop over and over again, the adversary's power is effectively limited to blocking votes it dislikes by stopping processors in between flipping their local coins to decide on the value of the votes and actually writing the votes out to the registers. The adversary's control is limited by running the protocol for long enough that the sum of these blocked votes is likely to be only a fraction of the total vote, a process that requires accumulating  $\Omega(n^2)$  votes.

In the original shared coin protocol of Aspnes and Herlihy [AH90a], each processor decides on a value when it sees a total vote whose absolute value is at least a constant multiple of  $n$  from the origin. For each of the expected  $\Theta(n^2)$  votes,  $\Theta(n^2)$  register operations are executed, giving a total running time of  $\Theta(n^4)$  operations. Unfortunately, both the implementation of the counter representing the position of the random walk and the mechanism for repeatedly running the shared coin require a potentially unbounded amount of space. This problem was corrected in a protocol of Attiya, Dolev, and Shavit [ADS89], which retained the multiple rounds of its predecessor but cleverly reused the space used by old shared coins once they were no longer needed.

A simpler descendent of the shared coin protocol of Aspnes and Herlihy, which also requires only bounded space, is the shared coin protocol described in Chapter 4. This protocol, by using a more sophisticated termination condition, guarantees that the processors *always* agree on its outcome. A simple modification of this protocol gives a consensus protocol that does not require multiple executions of a shared coin; which can be implemented using only three  $O(\log n)$ -bit counters, supporting increment, decrement, and read operations; and which runs in only  $\Theta(n^2)$  expected *counter* operations. However, this apparent speed is lost in the implementation of the counter, because  $\Theta(n^2)$  register operations are needed for each counter operation, giving it the same running time of  $\Theta(n^4)$  expected register operations as its predecessors. Since the consensus protocol of Chapter 4 first appeared [Asp90], other researchers [BR90, DHPW92] have described weaker primitives that act sufficiently like counters to make the protocol work and which use only a linear number of register operations for each counter operation. Using these primitives in place of the counters gives a consensus protocol that runs in

expected  $\Theta(n^3)$  register operations.

An alternative to having each processor finish the protocol when it sees a total vote far from the origin is to simply gather votes until some predetermined quorum is reached. The first shared coin protocol to use this technique is that of Saks, Shavit, and Woll [SSW91]. It is still necessary to gather  $\Omega(n^2)$  votes to overcome the effect of votes withheld by the scheduler, and in fact the Saks-Shavit-Woll protocol still requires  $\Theta(n^4)$  register operations. Furthermore, it is unlikely that any protocol that runs in a fixed number of total operations can guarantee that all processors agree on the outcome of the coin; thus it is necessary to retain the complex multiple rounds of the Aspnes-Herlihy protocol in some form. However, stopping the protocol after a specified number of votes are collected has a very important consequence: it is no longer necessary for a processor to check for termination after every vote it casts.

This remarkable fact was observed by Bracha and Rachman [BR91] and is the basis for their fast shared coin protocol. In this protocol, as in previous shared coin protocols, the processors repeatedly generate random  $\pm 1$  votes and add them to a running total. After a quorum of  $\Theta(n^2)$  votes are collected processors may decide on the output of the shared coin based on the sign of the total vote. But each processor only checks if the quorum has been reached after every  $O(n/\log n)$  votes— so the processors can generate an additional  $O(n^2/\log n)$  “extra” votes beyond the “common” votes making up the quorum. However, by making the number of common votes large enough compared to the number of extra votes, the probability that the extra votes will change the sign of the total vote can be made arbitrarily small. Thus, even if one processor reads the total vote immediately after the quorum is reached and another reads it after many extra votes have been cast, it is still likely that both will agree with each other on the outcome of the shared coin. In addition, because each processor only needs to compute the total vote once, after it has seen a full quorum, no counters or other complicated primitives are needed to keep track of the voting. Each processor simply maintains in its own register a tally of all the votes it has cast, and computes the total vote by summing the tallies in all of the registers. The result is that the protocol requires only  $O(n^2 \log n)$  expected total register operations.

There is, however, still room for improvement. All of the shared coin protocols we have described suffer from a fundamental flaw: if the scheduler stops all but one of the processors, that lone processor is still forced to

generate  $\Omega(n^2)$  local coin flips. The essence of wait-freeness is bounding the work done by a single processor, despite the failures of other processors. But the bound on the work done by a single processor, in every one of these protocols, is asymptotically no better than the bound on the work done by all of the processors together.

Chapter 5 shows that wait-free consensus can be achieved without forcing a fast processor to do most of the work. I describe a shared coin protocol in which the processors cast votes of steadily increasing weights. In effect, a fast processor or a processor running in isolation becomes “impatient” and starts casting large votes to finish the protocol more quickly. This mechanism does grant the adversary greater control, because it can choose from up to  $n$  different weights (one for each processor) when determining the weight of the next vote to be cast. One effect of this control is that a more sophisticated analysis is required than for the unweighted-voting protocols. Still, with appropriately-chosen parameters the protocol guarantees that each processor finishes after only  $O(n \log^2 n)$  expected operations.

The organization of the dissertation is as follows. Chapters 2 and 3 provide a framework of definitions for the material in the later chapters. Chapter 2 describes the asynchronous shared-memory model in detail and compares it with other models of distributed systems. Chapter 3 formally defines the consensus problem and its relationship to the problem of constructing shared coins. The main results appear in Chapters 4 and 5. Chapter 4 describes the simple consensus protocol based on a random walk. Chapter 5 describes the faster protocol based on weighted voting. Finally, Chapter 6 compares these results to other solutions to the problem of wait-free consensus and discusses possible directions for future work.

Much of the content of Chapters 4 and 5 also appears in [Asp90] and [AW92], respectively. Some of the material in Chapter 3 is derived from [AH90a].

# Chapter 2

## The Asynchronous Shared-Memory Model

This chapter gives a detailed description of the asynchronous shared-memory model. This model is the standard one for analyzing wait-free consensus protocols [Abr88, ADS89, AH90a, Asp90, AW92, BR90, BR91, DHPW92, SSW91]. Though it appears in varying guises, all are essentially equivalent. The description of the model here largely follows that of the “weak model” of Abrahamson [Abr88]. The reader interested in a more formal definition of the model may find one in [AH90a] based on the I/O Automaton model of Lynch [Lyn88].

### 2.1 Basic elements

The system consists of a collection of  $n$  **processors**, state machines whose behavior is typically specified by a high-level **protocol**. In principle no limits are assumed on the computational power of the processors, although in practice none of the protocols described in this document will require much local computation.

The processors can communicate only by executing **read** and **write** operations on a collection of **single-writer, multi-reader atomic registers** [Lam77, Lam86b]. Each of these registers is associated with one of the processors, its **owner**. Only the owner of a register is allowed to write to it, although any of the processors may read from it. **Atomicity** means that

read and write operations act as if they take place instantaneously: they never fail, and the result of concurrent execution of multiple operations on the same register is consistent with their having occurred sequentially.

The assumptions behind atomicity may appear to be rather strong, especially in a model that is designed to be as harsh as possible. However, it turns out that atomic registers are not powerful enough to implement deterministically such simple synchronization primitives as queues or test-and-set bits [Her91], and may be constructed efficiently from much weaker primitives in a variety of ways [BP87, IL87, NW87, Pet83, SAG87]. So in fact the apparent strength of atomic registers is somewhat illusory.

## 2.2 Time and asynchrony

The systems represented by the model may have many events occurring concurrently. However, because the only communication between processors in the system is by operations on atomic registers, it is possible to represent its behavior using a **global-time model** [BD88, Lam86a, Lam86b]. Instead of treating operations on the registers as occurring over possibly-overlapping intervals of time, they are treated as occurring instantaneously. The history of an execution of the system can thus be described simply as a sequence of operations. Concurrency in the system as a whole is modeled by the interleaving of operations from different processors in this sequence.

The actual order of the interleaving is the primary source of nondeterminism in the system. At any given time there may be up to  $n$  processors that are ready to execute another operation; how, then, does the system choose which of the processors will run next? We would like to make as few assumptions here as possible, so that our protocols will work under the widest possible set of circumstances. One way of doing this is to assign control over timing to an **adversary scheduler**, a function that chooses a processor to run at each step based on the previous history and current state of the system. The adversary scheduler is not bound by any fairness constraints; it may start and stop processors at will, doing whatever is necessary to prevent a protocol from executing correctly. In addition, no limits are placed on the scheduler's computational power or knowledge of the programming or internal states of the processors. However, its control is limited only to the timing of events in the system—it cannot, for example, cause a read operation to return the

wrong value or a processor to deviate from its programming.

The definition of a **wait-free** protocol implicitly depends on having such a powerful adversary. A protocol is said to be wait-free if every processor finishes the protocol in a finite number of its own steps, *regardless of the relative speeds of the other processors*. The adversary in the asynchronous shared-memory model simply represents the universal quantifier hidden in that condition. If we can design a protocol that will beat an all-powerful adversary, we will know that the protocol will succeed in the far easier task of working correctly in whatever circumstances chance and the workings of a real system might throw at it.

### 2.3 Randomization

In order to solve the consensus problem in the presence of an adversary scheduler, the processors will need to be able to act nondeterministically. In addition to giving each processor the ability to write to its own registers and to read from any of the registers, we will give each processor the ability to flip a **local coin**. This operation provides the processor with a random bit that cannot be predicted by the scheduler in advance, though it is known to the scheduler immediately afterwards by virtue of the scheduler's ability to see the internal states of the processors. The timing of coin-flip operations, like that of read and write operations, is under the control of the scheduler.

### 2.4 Relation to other models

There are other models that are closely related to the asynchronous shared-memory model. In particular it is tempting to define the property of being **wait-free** as the property that a protocol will finish (that is, one processor will finish) even in the presence of up to  $n - 1$  **halting failures**, where a halting failure is an event after which a processor executes no more operations. Such a definition would make wait-freeness a natural extension of **t-resilience**, the property of working in the presence of up to  $t$  halting failures.

However, in the context of a totally asynchronous system this definition is unnecessarily restrictive. It is true that the adversary is able to *simulate* up to  $n - 1$  halting failures simply by choosing not to run "halted" proces-

sors ever again. However, there is no reason to believe that dead processors are the only source of difficulty in an asynchronous environment. For example, the adversary could choose to put some processor to sleep for a very long interval, waking it only when its view of the world was so outdated that its misguided actions would only hinder the completion of a protocol. As the hallway example in the introduction shows, stopping a processor and reawakening it much later can be even more devastating than stopping a processor forever. Furthermore, distinguishing between slow processors and dead ones requires either an assumption that slow processors must take a step after some bounded interval, or that fast processors may execute a potentially unbounded number of operations waiting for the slow processors to revive. The first assumption imposes a weak form of synchrony on the system, violating the principle of avoiding helpful assumptions; the second makes it difficult to measure the efficiency of a protocol. For these reasons we avoid the issue completely by using the more general definition.

Other alternatives to the model involve changing the underlying communications medium from atomic registers, either by adopting stronger primitives that provide greater synchronization, or by moving to some sort of message-passing model. We avoid the first approach because, as always, we would like to work in as weak a model as possible. However, the question of how a different choice of primitives can affect the difficulty of solving wait-free consensus is an interesting one about which little is known, except for the deterministic case [LAA87, Her91].

Moving to a message-passing model presents new difficulties. In general, the defining property of a message-passing model is that the processors communicate by sending messages to each other directly, rather than operating on a common pool of registers or other primitives. Message-passing models come in bewildering variety; a general taxonomy can be found in [LL90]. Dolev et al. [DDS87] classify a large collection of message-passing models and show which are capable of solving consensus deterministically.

Among these many models, one has traditionally been associated with solving asynchronous consensus [BND89, BT83, CM89, FLP85]. In this model, the adversary is allowed to (i) stop up to  $t$  processors and (ii) delay messages arbitrarily. Unfortunately, a simple partition argument shows that in this model one cannot solve consensus even with a randomized algorithm if at least  $n/2$  processors can fail [BT83]. Intuitively, the adversary can divide the processors into two groups of size  $n/2$  and delay all messages

passing between the groups. As neither group will be able to distinguish this partitioning from the other group actually being dead, the two groups will independently come up with decisions that may be inconsistent. On the other hand, solutions to the wait-free consensus problem for shared memory can be used to obtain solutions to consensus problems for message-passing models with weaker failure conditions by simulating the shared memory. An example of this technique may be found in [BND89]. A general comparison of the power of shared-memory and message-passing models in the presence of halting failures can be found in [CM89].

## 2.5 Performance measures

It is not immediately obvious how best to measure the performance of a wait-free decision protocol. Two measures are very natural for the asynchronous model, as they impose no implicit assumptions on the scheduling of operations in the system. These are the **total work** measure, which simply counts the total number of register operations executed by all the processors together until every processor has finished the protocol; and the **per-processor work** measure, which takes the maximum over all processors of the number of register operations executed by each processor individually before it finishes the protocol.

The per-processor measure is closer to the spirit of the wait-free guarantee that each processor finishes in a finite number of its own steps, as it gives an upper bound on what that finite number is. However, prior to the protocol of Chapter 5, for every known consensus protocol (see Table 6.1) the two measures were within a constant factor of each other. As a result only the total work measure has typically been considered. This usage is in contrast to what is needed in situations where processors may re-enter the protocol repeatedly, as in protocols for simulating various shared abstract data types [AAD<sup>+</sup>90, AG91, AH90b, And90, DHPW92, Her91, Plo89] or for timestamping and similar mechanisms [DS89, DW92, IL87]. In these protocols one is typically interested in the number of register operations each processor must execute to simulate a single operation on the shared object, an inherently per-processor measure, and the total work is meaningful only when interpreted in an amortized sense.

Given the usefulness of the per-processor measure in this broader con-

text, I will concentrate primarily on it. However, because the total-work measure has traditionally been used to analyze consensus protocols it will be considered as well.

An alternative to these measures that has seen some use in analyzing wait-free protocols is the **rounds** measure of asynchronous time [AFL83, ALS90, LF81, SSW91]. It is used for models that represent halting failures explicitly. When using this measure, up to  $n - 1$  processes may be designated as faulty at the discretion of the adversary; once a processor becomes faulty it is never allowed to execute another operation. A round is a minimal interval during which every non-faulty processor executes at least one operation. The measure is simply the number of these rounds. In effect, this measure counts the operations of the slowest non-faulty processor at any given point in the execution. If a slow processor executes only one operation in a given interval, only one round has elapsed, even though a faster processor might have carried out hundreds of operations during the same interval.

The rounds measure is reasonable if one defines the property of being wait-free as equivalent to being able to survive up to  $n - 1$  halting failures. However, as explained above, in the context of a totally asynchronous system this definition is unnecessarily restrictive. But once we adopt the more general definitions we quickly run into trouble. If some processor stops and then starts again much later during the execution of the protocol, the entire period that the processor is inactive counts as only one round. As a result the rounds measure implicitly resolves the problem of distinguishing slow processors from dead ones by guaranteeing that processors will either run at bounded relative speeds or not run at all. This is in conflict with the goal of using a model that is as general as possible, and for this reason the rounds measure will not be used here.<sup>1</sup>

---

<sup>1</sup>A notion of "rounds" does appear in Section 3.3; these rounds are part of the internal structure of the protocol described there and have no relation to the rounds measure.

# Chapter 3

## Consensus and Shared Coins

This chapter formally describes the problem of solving consensus and the closely-related problem of construction a shared coin, and gives an example of a method for solving consensus using a shared coin. This last technique will be of particular importance in Chapter 5.

### 3.1 Consensus

**Consensus** is a decision problem in which  $n$  processors, each starting with a value (0 or 1) not known to the others, must collectively agree on a single value. A **consensus protocol** is a distributed protocol for solving consensus. It is **correct** if it meets the following conditions [CIL87]:

- **Consistency.** All processors decide on the same value.
- **Termination.** Every processor decides on some value in finite expected time.
- **Validity.** If every processor starts with the same value, every processor decides on that value.

The basic idea behind consensus is to allow the processors to make a collective decision. For this purpose, the consistency condition is the most fundamental of the correctness conditions, as it is what actually guarantees that the processors agree. The termination condition is phrased to apply in

many possible models; in the asynchronous shared-memory model it translates into requiring that the protocol be **wait-free**, as it requires that processors must finish in finite expected time regardless of the actions of the adversary scheduler.

If it happens that the processors already agree with each other, we want the consensus protocol to ratify that agreement rather than veto it; hence the validity condition. From a less practical perspective the validity condition is needed because its absence makes the problem uninteresting, since all of the processors could just decide 0 every time the protocol is run without any communication at all.

If we are allowed to make convenient assumptions about the system, consensus is not a difficult problem. For example, on a PRAM (perhaps the friendliest cousin of asynchronous shared-memory) consensus reduces to simply taking any function we like of the input values that satisfies the validity condition. In general, in any model where both the processors and the communications medium are reliable the problem can be solved simply by having the processors exchange information about their inputs until all of them know the entire set of inputs; at this point each can individually compute a function of the inputs as in the PRAM case to come up with the decision value for the protocol. It is only when we move to a model, like asynchronous shared-memory, that allows processors to fail that consensus becomes hard.

One difficulty is that the harsh assumptions of the asynchronous shared-memory model can amplify the correctness conditions in ways that may not be immediately obvious. For example, the validity conditions implies that the adversary can always force the processors to decide on a particular value by running only those processors that started with that value. Because these “live” processors are unable to see the differing input values of the “dead” processors, they will see a situation indistinguishable from one in which every processor started with the same value. In this latter case, the validity condition would force the processors to decide on that common value. So because of their limited knowledge, the live processors must decide on the only input value they can see, even though there may be other processors that disagree with it. This example shows that one must be very careful about what assumptions one makes in the model, as they can subtly affect what a protocol is allowed to do.

## 3.2 Shared coins

In order to solve the consensus problem we will need to cope with the considerable power of the adversary. We cannot modify the model to place restrictions on the adversary; instead, we must find some way of getting the processors to reach agreement in spite of the adversary's interference.

One way is to base a consensus protocol on a stronger primitive, the **shared coin**. A shared coin is a decision protocol in which each processor decides on a bit, which with some probability  $\delta$  will be the same value that every other processor decides on. But unlike consensus, the actual value chosen will *not* always be under the control of the adversary. In order to prevent this control, given the adversary's ability to run only some of the processors, we must drop the validity condition and with it the notion of input bits. What we are left with is the following definition.

A **shared coin protocol with agreement parameter**<sup>1</sup>  $\delta$  is a distributed decision protocol that satisfies these two conditions:

- **Termination.** Every processor decides on some value in finite expected time.
- **Probabilistic agreement.** For each value  $b$  (0 or 1), the probability that every process decides on  $b$  is at least  $\delta$ .

The probabilistic agreement condition guarantees that with probability  $2\delta$  the outcome of the shared coin protocol is agreed on by all processors and is indistinguishable from flipping a fair coin. With probability  $1 - 2\delta$ , no guarantees whatsoever are made; it is possible that the processors will not agree with each other at all, or that the adversary will be able to choose what value each processor decides on. Some sort of adversary control is always possible, as it is known that a wait-free shared coin with  $\delta$  exactly equal to  $1/2$  is impossible [AH90a].

The agreement parameter is not the only possible parameter for shared coin, merely the one that is most convenient when building consensus protocols. If we wish to use the coin directly (for example, as a source of semi-random bits [SV86] in a distributed algorithm) a more natural parameter

---

<sup>1</sup>Called the **defiance parameter** in [AH90a]. The less melodramatic term **agreement parameter** is taken from [SSW91].

is the **bias**,  $\epsilon$ , defined by  $\epsilon = 1/2 - \delta$ . In terms of the bias the agreement property can be restated as follows:

- **Bounded bias.** The probability that at least one processor decides on a given value is at most  $1/2 + \epsilon$ .

This property says in effect that the adversary can force some processor to see a particular outcome with only  $\epsilon$  greater probability than if the processors were actually collectively flipping a fair coin.

In some circumstances we would like to guarantee that all of the processors *always* agree on the outcome of the coin, even though the adversary might have been able to control what that outcome is. A shared coin that guarantees agreement will be called **robust**. As will be seen in Chapter 4, robust shared coins can often be converted directly into consensus protocols by the addition of only a small amount of machinery. However Chapter 5 describes an intrinsically non-robust shared coin; in this situation more sophisticated techniques are needed to achieve consensus. One approach is described in the next section.

### 3.3 Consensus using shared coins

It is a well-established result that one can construct a consensus protocol from a shared coin with constant agreement parameter [ADS89, AH90a, SSW91]. This section gives as an example the first of these constructions [AH90a]. As we shall see, this construction gives a consensus protocol which requires an expected  $O((T(n) + n)/\delta)$  operations per processor and  $O((T'(n) + n^2)/\delta)$  total operations, where  $T(n)$  and  $T'(n)$  are the expected number of operations per processor and total operations for the shared coin protocol.

Pseudocode for each processor's behavior in the shared-coin-based consensus protocol is given in Figure 3.1. Each processor has a register of its own with two fields: *prefer* and *round*, initialized to  $(\perp, 0)$ . In addition there are assumed to be a (potentially unbounded) collection of shared coin primitives, one for each "round" of the protocol. Two special terms are used to simplify the description of the protocol. A processor is a **leader** if its *round* field is greater than or equal to every other process's *round* field. Two processors **agree** if both their *prefer* fields are equal, and neither field is  $\perp$ .

```

1 procedure consensus(input)
2 (prefer, round)  $\leftarrow$  (input, 1)
3 repeat
4     read all the registers
5     if all who disagree trail by 2 and I'm a leader then
6         output prefer
7     else if leaders agree then
8         (prefer, round)  $\leftarrow$  (leader preference, round + 1)
9     else if prefer  $\neq \perp$  then
10        (prefer, round)  $\leftarrow$  ( $\perp$ , round)
11     else
12        (prefer, round)  $\leftarrow$  (shared_coin[round], round + 1)

```

Figure 3.1: Consensus from a shared coin.

Let us sketch out the workings of the protocol. The most serious problem that the protocol is designed to solve is how to neutralize “slow” processors that have old, out-of-date views of the world. Because such processors end up with low *round* values relative to the “fast” processors, they are effectively excluded from the real decision-making in the protocol until they manage to catch up to their faster comrades.

Intuitively, the decision-making process consists of the leaders running the shared coin protocol in line 12. It is not necessarily the case that all of the leaders at each round will take part in the shared coin protocol, as those that arrive earliest may not see disagreement and will execute line 8 instead. However, those early arrivals must in fact agree with each other, and so with probability at least  $\delta$  the others will switch to agree with them at any given round. It follows that the expected number of rounds until agreement is  $O(1/\delta)$ .

Once the leaders agree, the slower processors are forced to adopt the leaders’ position by executing line 8. The protocol terminates when the agreeing processors advance far enough (2 rounds) to know that any processor that disagrees will pass through line 8 before catching up and becoming a leader itself.

This explanation is informal, and glosses over many important but tedious

details of the protocol. The interested reader is referred to [AH90a] for a more thorough description of the construction including a full proof of correctness. Alternative constructions with similar performance may be found in [ADS89] and [SSW91].

For our purposes it will suffice to summarize the relevant results from [AH90a]:

**Theorem 3.1 ([AH90a])** *The protocol of Figure 3.1 implements a consensus protocol that requires an expected  $O(1/\delta)$  rounds, where  $\delta$  is the agreement parameter of the shared coin.*

From which it follows that:

**Corollary 3.2** *The protocol of Figure 3.1 implements a consensus protocol that requires an expected  $O((T(n) + n)/\delta)$  operations per processor and  $O((T'(n) + n^2)/\delta)$  operations in total; where  $\delta$  is the agreement parameter,  $T(n)$  the expected number of operations per processor, and  $T'(n)$  the expected number of operations in total for the shared coin.*

**Proof:** From the theorem, we expect at most  $O(1/\delta)$  rounds.

In each round, each processor executes at most  $2n$  read operations, one instance of the shared coin, and two write operations, for a total of  $2n + 2 + T(n)$  operations.

Similarly, in each round the processors collectively execute at most  $2n^2$  read operations,  $2n$  write operations, and one instance of the shared coin, for a total of  $2n^2 + 2n + T'(n)$  operations. ■

## Chapter 4

# Consensus Using a Random Walk

All currently-known wait-free consensus protocols that run in polynomial time are based on some form of a shared coin protocol. The key insight used in constructing shared coins is that it is dangerous to give too much power over the outcome of the protocol to any one processor at any given time. Such tyranny, like all tyrannies, runs the risk of a sudden change in policy following an assassination, and thereby gives control over policy to potential assassins like our adversary scheduler. In all currently known shared coin protocols each individual processor's power is minimized by having the processors repeatedly cast small random votes for the two decision values.

How this voting process is best represented depends on the method used to decide when it is finished. In this chapter we describe a shared coin and a consensus protocol in which the voting ends when the difference between the number of 1 votes and 0 votes is large. Under such circumstances the voting process can be viewed as a random walk in which each vote moves the total up or down by one until an absorbing barrier at  $\pm K$  is reached (where  $K$  is a parameter of the protocol). In fact, the original polynomial-time shared coin of [AH90a] worked on exactly this principle. Unfortunately, a simple implementation of a random walk does not guarantee agreement, as the adversary can allow one processor to see a total greater than  $K$  and decide 1, and then, by releasing negative votes "trapped" inside stopped processors, move the total down out of the decision range so that with some nonzero probability the other processors will eventually move it to  $-K$  and

decide 0. So to use a simple random-walk-based shared coin in a consensus protocol one would need to run it repeatedly as described in Section 3.3.

The protocols described in this chapter avoid the need for such methods by extending the random walk to incorporate the function of detecting agreement. As a result we obtain a *robust* shared coin, described in Section 4.2, which guarantees that all processors agree on its outcome. Because the coin guarantees agreement, it can be modified in to obtain a consensus protocol simply by attaching a preamble to ensure validity, as described in Section 4.4. The resulting consensus protocol (and its variants, obtained by replacing the counter implementation [BR90, DHPW92]) are particularly simple, as they are the only known wait-free consensus protocols that do not require the repeated execution of a non-robust shared coin protocol and the multi-round superstructure that comes with it.

The simplicity of the protocol also allows some optimizations that are more difficult when using a non-robust coin. The consensus protocol is designed to require fewer total operations if fewer processors actually participate in it, a feature which becomes important when, for example, the protocol is used as a primitive for building shared data structures which only a few processors might attempt to access simultaneously.

The chapter is organized as follows. Section 4.1 describes some properties of random walks that will be used later in the chapter. Section 4.2 describes the robust shared coin protocol and proves its correctness. The description of the robust shared coin protocol assumes the presence of an **atomic counter**, providing increment, decrement, and read operations that appear to occur sequentially; Section 4.3 shows how such a counter may be built from single-writer atomic registers at the cost of  $O(n^2)$  register operations per counter operation. Finally, Section 4.4 describes the consensus protocol obtained by modifying the robust shared coin.

## 4.1 Random walks

Let us begin by stating a few basic lemmas about the behavior of random walks.

**Lemma 4.1** *Consider a symmetric random walk with step size 1 running between absorbing barriers at  $a$  and  $b$  and starting at  $x$ , where  $a < x < b$ . Then:*

1. The expected number of steps until one of the barriers is reached is given by  $(x - a)(b - x)$ , which is always less than or equal to  $(\frac{b-a}{2})^2$ .
2. The probability that the random walk hits  $b$  before  $a$  is  $\frac{x-a}{b-a}$ .

**Proof:** The random walk described is just a form of the classical gambler's ruin problem. See [Fel68, pp. 344-349]. ■

**Lemma 4.2** Consider a symmetric random walk with step size 1 running between a reflecting barrier at  $a$  and an absorbing barrier at  $b$ , starting at position  $x$ ,  $a < x < b$ . Then the expected time until  $b$  is reached is  $(x - (a - (b - a)))(b - x) \leq (b - a)^2$

**Proof:** This random walk can be obtained from the random walk with absorbing barriers at  $b$  and  $a - (b - a)$  by the transformation  $x \mapsto a + |x - a|$ . ■

The following critical lemma describes a modified random walk that will be of great importance in analyzing the shared coin and consensus protocols:

**Lemma 4.3** Consider a symmetric random walk with absorbing barriers at  $a$  and  $b$  with the following twist: a point  $c$ ,  $a < c < b$  is chosen as the center of the random walk. The adversary chooses the starting position  $x$  of the random walk to be anywhere in the range from  $a$  to  $b$ . Also, before each step, the adversary may choose between moving randomly in either direction with probability  $1/2$ , or moving away from  $c$  with probability  $1$ . No matter what choices the adversary makes, the expected number of steps until one of the barriers is reached is at most  $(b - a)^2$

**Proof:** The game described can be thought of as a controlled Markov process [DY75] in which the adversary is trying to maximize the expected time. Because this process is played over a finite set of states, a standard result of the theory of controlled Markov processes can be applied. This result states that the maximum time can be achieved by an adversary using a simple strategy, one which chooses the same option from each state at all times.

Such a strategy can be specified by listing the points where the adversary chooses to force the particle to move away from  $c$ . We can think of these

points as dividing the range of the random walk into intervals; between each pair of points where the adversary forces the particle to move deterministically is a region where the particle moves randomly. The points at the edge of these random regions act like barriers in a random walk. A point on the side away from  $c$  pushes the particle into a new region and so acts like an absorbing barrier, while a point on the side toward  $c$  pushes the particle back into the old region and so acts like a reflecting barrier. Thus the region containing  $c$  acts like a random walk with two absorbing barriers, and the remaining regions act like random walks with one absorbing barrier (on the side away from  $c$ ) and one reflecting barrier (on the side toward  $c$ ).

Because each barrier can only be crossed away from  $c$ , once the particle leaves a region it can never return. Now, suppose the particle starts in a region with width  $w_1$ . After at most  $w_1^2$  steps on average (by Lemmas 4.1 or 4.2) it will pass into a new region of width  $w_2$ ; after an additional  $w_2^2$  steps it will pass into a new region of width  $w_3$ , and so on until either  $a$  or  $b$  is reached. Since these regions all fit between  $a$  and  $b$ ,  $\sum w_i \leq b - a$ , and thus (since each  $w_i > 0$ )  $\sum w_i^2 \leq (b - a)^2$ . ■

Though the bound in Lemma 4.3 is proved for the case of a very powerful adversary that is always allowed to choose between a random move and a deterministic move at each step, the bound applies equally well to a weaker adversary whose choices are more constrained, as the stronger adversary could always choose to operate within the weaker adversary's constraints. This technique, of proving bounds for a strong adversary that carry over to a weaker one, has great simplifying power. It will be used extensively in the analysis of the shared coin and consensus protocols.

## 4.2 The robust shared coin protocol

Figure 4.1 shows pseudocode for each processor's behavior in the robust shared coin protocol. The coin is constructed using an **atomic counter**, which supports atomic increment, decrement, and read operations. In this section, these operations are assumed to take unit time. The counter is initialized to 0. The processor's local coin is represented by the procedure *local\_flip*, which returns the values  $-1$  and  $1$  with equal probability.

A processor's behavior in the protocol is represented in pictorial form in Figure 4.2. While a processor reads values in the central range from  $-K$

Shared data:

counter *counter* with range  $[-K - 3n, K + 3n]$ , initialized to 0

```
1 procedure shared_coin()
2 repeat
3     c  $\leftarrow$  counter
4     if c  $\leq$   $-(K + n)$  then output 0
5     else if c  $\geq$   $(K + n)$  then output 1
6     else if c  $\leq$   $-K$  then decrement counter
7     else if c  $\geq$   $K$  then increment counter
8     else
9         if local_flip() = 1 then increment counter
10        else decrement counter
```

Figure 4.1: Robust shared coin protocol.

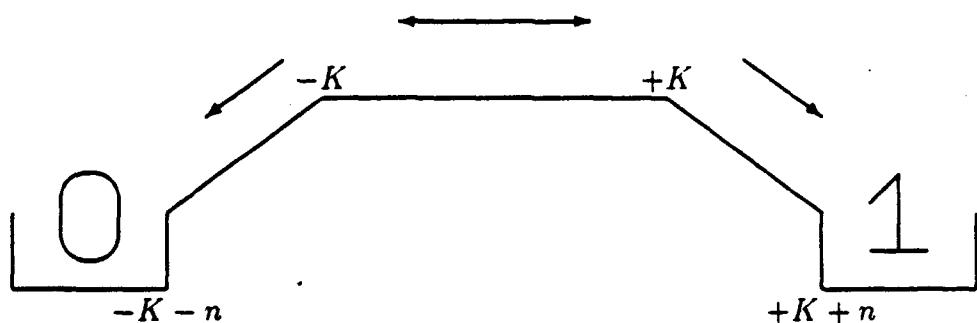


Figure 4.2: Pictorial representation of robust shared coin protocol.

to  $K$  (where  $K$  is a parameter of the protocol) it flips a local fair coin to decide whether to increment or decrement the counter. This part of the protocol is essentially the same as the random-walk-based shared coin of Aspnes and Herlihy [AH90a]. What is new is the addition of a “slope” at either side of the random walk. On these slopes, a processor does not move the counter randomly but instead always moves it away from the center. When a processor reads a counter value in one of the “buckets” beyond the slopes, it decides either 0 or 1 depending on the sign of the counter.

If the slopes are wide enough, once any processor has seen a value that causes it to decide, all other processors will see values that cause them to push the counter toward the same decision. This mechanism eliminates the possibility that delayed writes might move the counter out of the decision range and allow the random walk (with small but non-negligible probability) to wander over to the other side. More formally, we can show:

**Lemma 4.4** *If any processor reads a counter value  $v \geq (K + n)$ , then all subsequent reads will return values greater than or equal to  $K + 1$ ; in the symmetric case where  $v \leq -(K + n)$ , all subsequent values read will be less than or equal to  $-(K + 1)$ .*

**Proof:** Suppose that a processor has read  $v \geq (K + n)$ ; then it immediately terminates leaving  $n - 1$  running processors. Thus the number  $d$  of processors that will execute a decrement before their next read is at most  $n - 1$ . Let  $l = c - d$  where  $c$  is the value stored in the counter. Since  $c \geq (K + n)$ , it must be the case that  $l \geq K + 1$ . Now consider the effect of the actions the scheduler can take. If it allows a decrement to proceed,  $c$  and  $d$  both drop by 1 and  $l$  remains constant. If it allows an increment to occur,  $c$  increases and  $l$  increases with it. If it allows a read, the value read is  $c \geq l \geq K + 1$ , and thus  $d$  is unaffected. In each case  $l$  remains at least  $K + 1$ , and the claim follows since  $c \geq l$ . The proof of the symmetric case is similar. ■

The consistency property follows immediately from Lemma 4.4. A similar argument shows that the counter will not overflow:

**Lemma 4.5** *The counter value never leaves the range  $[K - 3n, K + 3n]$  in any execution of the shared coin protocol.*

**Proof:** Suppose that the counter reaches  $K + 2n$  at some point. Then each processor will execute at most one increment or decrement operation before

it reads the counter, at which point it will decide 1 and execute no additional operations. Thus the counter cannot exceed  $K + 2n + n = K + 3n$ . The full result follows by symmetry. ■

Proving the termination and bounded bias properties of the shared coin requires some additional machinery. Define the **true position**  $t$  of the random walk to be the value in the counter, plus 1 for each processor that will increment the counter before its next read, and minus 1 for each processor that will decrement the counter before its next read. The following Lemma relates the value read by a processor to the true position of the random walk:

**Lemma 4.6** *Let  $c$  be a value read from the counter by some processor and  $t$  the true position of the random walk in the state preceding the read. Then  $|c - t| \leq n - 1$ .*

**Proof:** There can be at most  $n - 1$  processors with pending increments or decrements. ■

Let us assume hereafter that the scheduler can cause a processor to read any value between  $t - (n - 1)$  and  $t + (n - 1)$ . Because such a scheduler could always choose to simulate any scheduler the protocol will actually face, any “good” statement we can prove with the assumption will carry over to the situation without it. The advantage of granting the adversary this additional power is that it allows us to forget about the vagaries of the counter value. Instead we can treat the protocol as a controlled random walk using the true position  $t$ .

Consider the lower part of Figure 4.3 (the upper part simply repeats Figure 4.2 without the buckets.) If the true position  $t$  is in the central region between  $-K + (n - 1)$  and  $K - (n - 1)$ , then Lemma 4.6 implies that any processor that reads the counter will see a value between  $-K$  and  $K$  and move  $t$  randomly. In the two immediately adjacent regions, any processor will either read a value between  $-K$  and  $K$ , and move  $t$  randomly, or read a value that causes it to move  $t$  away from 0. Finally, any processor that reads a value in the outermost regions where  $|t| > K + (n - 1)$  will either make a decision or move  $t$  away from 0. In each of these cases, the scheduler is never allowed to force that true position toward 0; and if  $K$  is large relative to  $n$  much of the execution of the protocol will be spent in the central region where the scheduler’s control is ineffective. These two properties of the protocol are the basis of the proof of its termination and bounded bias, as shown below.

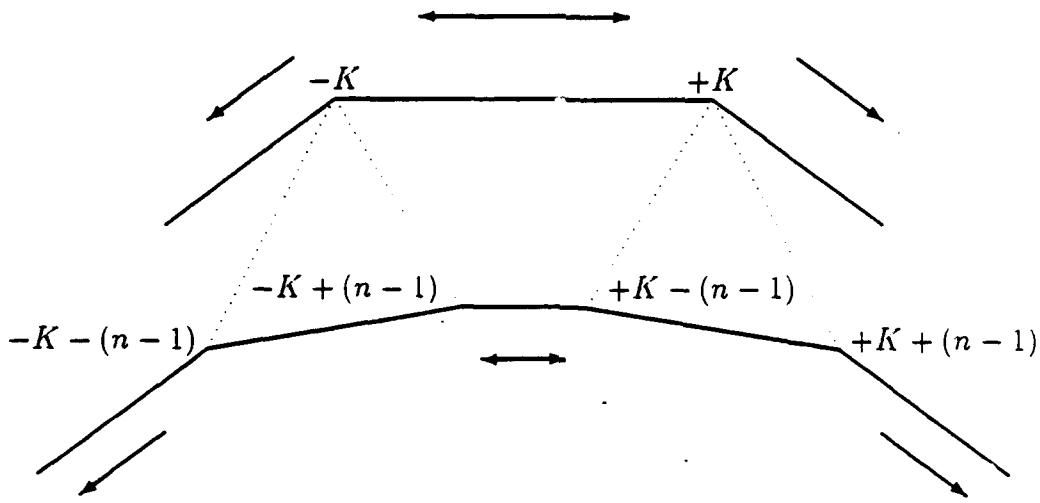


Figure 4.3: The protocol as a controlled random walk.

**Lemma 4.7** *The robust shared coin protocol executes an expected  $O((K + n)^2)$  total counter operations when  $K \geq n$ .*

**Proof:** If we consider the true position  $t$ , Lemma 4.6 implies that the scheduler can only force  $t$  up if  $t \geq K - (n - 1) \geq 1$  and down if  $t \leq -K + (n - 1) \leq -1$ . Furthermore if  $|t|$  ever exceeds  $K + n + (n - 1)$ , each processor will decide after its next read. Thus the movement of the true position is a controlled random walk in the sense of Lemma 4.3 with center 0 and barriers at  $\pm(K + 2n - 1)$ . The expected number of steps until a barrier is reached is at most  $4(K + 2n - 1)^2$  steps, which will be followed by at most  $2n$  operations as the processors each decide. Since each step takes a constant number of counter operations the expected number of operations required is  $O((K + n)^2)$ . ■

The time bound of Lemma 4.7 shows that every processor terminates in finite expected time when  $K \geq n$ . The bounded bias property is a consequence of the following lemma:

**Lemma 4.8** *Against any scheduler, the probability that the processors in the robust shared coin protocol will decide 1 is between  $\frac{K-(n-1)}{2K}$  and  $\frac{K+(n-1)}{2K}$ .*

**Proof:** Suppose the scheduler is trying to maximize the probability of deciding 1. Under the simplifying assumption it can force a decision of 1 as soon as  $t = K - (n - 1)$ ; however, if it allows  $t$  to slip below  $-K - (n - 1)$  the processors will eventually decide 0. When  $-K - (n - 1) < t \leq K - (n - 1)$  the scheduler may choose between moving  $t$  randomly or forcing  $t$  toward  $-K - (n - 1)$ . Clearly, forcing the counter toward  $-K - (n - 1)$  can only increase the probability of deciding 0, so choosing to move  $t$  randomly maximizes the probability of deciding 1. But if the scheduler makes this choice, the movement of the true position becomes a simple random walk with absorbing barriers at  $-K - (n - 1)$  and  $K - (n - 1)$ . By Lemma 4.1, the probability that  $t$  reaches  $K - (n - 1)$  first is  $\frac{K+(n-1)}{2K}$ . The lower bound follows by symmetry. ■

Combining the lemmas we obtain:

**Theorem 4.9** *When  $K > n$ , the protocol of Figure 4.1 implements a robust shared coin.*

**Proof:** Consistency follows from Lemma 4.4, termination from Lemma 4.7, and bounded bias from Lemma 4.8. ■

Lemma 4.8 allows  $K$  to be chosen to obtain arbitrarily small non-negative bias. Let the bias of the shared coin be  $\frac{1}{2} + \epsilon$ , then

$$\epsilon \leq \frac{n-1}{2K}$$

which gives

$$K \geq \frac{n-1}{2\epsilon}$$

Combining this inequality with Lemma 4.7 gives a bound on the worst-case expected running time for the protocol of  $O((n/\epsilon)^2)$  total counter operations. This time is comparable to the worst-case expected running times of the protocol's non-robust ancestors. The protocol thus achieves robustness without paying a significant cost in speed.

### 4.3 Implementing a bounded counter with atomic registers

The robust shared coin protocol assumes the presence of a shared counter supporting atomic increment, decrement, and read operations, with the restriction that no operation will be applied that will move the counter out of some fixed range  $[-r, r]$ . In practice such a counter is not likely to be available as a hardware primitive. Fortunately it is not difficult to implement a shared counter using atomic registers. However, some care must be taken to guarantee that the counter uses only a bounded amount of space.

Both Aspnes and Herlihy [AH90a] and Attiya, Dolev, and Shavit [ADS89] describe shared counter implementations. The two counter implementations both assign a register to hold the net increment due to each processor, so that the counter's value is simply the sum of the values in these registers. Both algorithms use simple atomic snapshot protocols to allow the entire set of registers to be read in a single atomic action.

Alas, neither implementation does quite what we would like. Even though the value stored in the counter will never exceed the range  $[-r, r]$ , the net increment due to an individual processor is potentially unbounded. The Aspnes-Herlihy protocol ignores this difficulty by assuming the presence of unbounded registers (which it also uses to implement the atomic scan.) The Attiya-Dolev-Shavit protocol uses only bounded registers, but enforces the bounds by prematurely terminating the shared coin protocol if any processor's register wanders out of a limited range. This premature termination occurs infrequently, and is acceptable in a shared coin that does not need to guarantee consistency. But it is not acceptable for a robust coin, as it may allow the scheduler to force some processor to choose one value (through premature termination) after another has already chosen a different value (through the normal workings of the shared coin protocol.)

A simple alternative to premature termination that still allows the size of the registers to be bounded is to store the remainder of each processor's contribution relative to some convenient modulus  $m$  greater than the total range  $2r + 1$ . The counter value can then be reconstructed as the unique  $v$  in the range  $[-r, r]$  that is congruent to the sum of the registers, modulo  $m$ . Pseudocode for the three counter operations using this technique is shown in Figure 4.4; it assumes the presence of an array of registers which can be

Shared data:

scannable array  $count[0 \dots n - 1]$ , initialized to 0

**procedure** *increment()*

$v \leftarrow count[me]$

$count[me] \leftarrow (v + 1) \bmod m$

**procedure** *decrement()*

$v \leftarrow count[me]$

$count[me] \leftarrow (v - 1) \bmod m$

**procedure** *read()*

scan  $count$

$v \leftarrow \sum_{i=0}^{n-1} count[i]$

**return**  $v'$  where  $-r \leq v' \leq r$  and  $v' \equiv v \pmod{m}$

Figure 4.4: Pseudocode for counter operations.

read in a single operation. Such an array can be simulated using an **atomic snapshot** protocol [AAD<sup>+</sup>90, AII90b, And90]. An atomic snapshot is an operation that returns a picture of the values in all of the registers in the array that is consistent with other pictures returned by other snapshots and with the order of non-overlapping write operations even though it may not correspond to the actual values in the registers at a particular point in time. Typically, it is necessary to make writes to registers in the scannable array be more than just simple writes to individual registers, so taking a snapshot of the array and writing to an element of the array will both be expensive. However, the algorithm of Afek et al. [AAD<sup>+</sup>90] allows an atomic scan operation to be implemented with  $O(n^2)$  bits of extra space and a maximum of  $O(n^2)$  primitive read and write operations for each snapshot and each write to a simulated register in the scannable array. Using their algorithm, 4.4 implements an atomic counter where each counter operations costs  $O(n^2)$  register operations.

## 4.4 The randomized consensus protocol

Figure 4.5 shows pseudocode for each processor's behavior in the randomized consensus protocol. The protocol uses three shared counters. The first two maintain a total of the number of participating processors that started with each of the inputs 0 and 1. The last is used as the counter for a modified version of the robust shared coin protocol. All of the counters have an initial value of 0.

The protocol is optimized for the case where few processors participate. We will define a processor to be **active** if it takes at least one step before some processor decides on a value, and denote by  $p$  the total number of active processors in a given execution. The protocol uses the counters  $a_0$  and  $a_1$  to keep track of the number of active processors by having each processor increment one or the other of these counters as it starts the protocol.

The protocol depends on being able to take an atomic snapshot of the counters. Since the first two counters are never decremented, such a snapshot can be obtained as described in Figure 4.6. Though the operation defined there is not wait-free, because it will not finish if  $a_0$  or  $a_1$  changes during some pass through the loop, this event can occur at most  $p$  times during any execution of the consensus protocol. So in fact the time to carry out the atomic snapshot will be bounded in the context in which it is used.

If the counters are not primitives but are instead constructed as described in Section 4.3 using an atomic scan operation, the overhead of Figure 4.6 can be avoided completely by simply reading all three counters in a single atomic scan of the arrays that implement them.

Several features of the protocol are worth noting. First of all, the same "slopes" that ensured consistency for the robust shared coin ensure consistency for the consensus protocol, for the same reasons. Second, the counters  $a_0$  and  $a_1$  allow the protocol to guarantee validity, as the random walk is only invoked if both have non-zero values. These counters are also used to minimize the range of the random walk, by taking advantage of the fact stated in the following lemma, a modification of Lemma 4.6:

**Lemma 4.10** *Let  $a_0, a_1, c$  be the values read from the counters by some processor and  $t$  the true position of the random walk in the state preceding the read. Then  $|c - t| \leq a_0 + a_1 - 1$ .*

**Proof:** There are at most  $a_0 + a_1 - 1$  processors with pending increments or

Shared data:

counter  $a_0$  with range  $[0, n]$ , initialized to 0  
counter  $a_1$  with range  $[0, n]$ , initialized to 0  
counter  $c$  with range  $[-4n, 4n]$ , initialized to 0

```
1 procedure consensus(input)
2 increment  $a_{input}$ 
3 repeat
4   read  $a_0, a_1, c$ 
5   if  $c \leq -2n$  then output 0
6   else if  $c \geq 2n$  then output 1
7   else if  $c \leq -(a_0 + a_1)$  or  $a_1 = 0$  then decrement  $c$ 
8   else if  $c \geq (a_0 + a_1)$  or  $a_0 = 0$  then increment  $c$ 
9   else
10      if local_flip() = 1 then increment  $c$ 
11      else decrement  $c$ 
```

Figure 4.5: Consensus protocol.

```
procedure scan_counters()
repeat
   $a_0 \leftarrow \text{read}(a_0)$ 
   $a_1 \leftarrow \text{read}(a_1)$ 
   $c \leftarrow \text{read}(c)$ 
   $a'_0 \leftarrow \text{read}(a_0)$ 
   $a'_1 \leftarrow \text{read}(a_1)$ 
until  $a'_0 = a_0$  and  $a'_1 = a_1$ 
return  $a_0, a_1, c$ 
```

Figure 4.6: Counter scan for randomized consensus protocol.

decrements. ■

To prove that the consensus protocol is correct, we must establish that it is consistent, that it terminates, and that it is valid. The proof of consistency is a straightforward modification of the proof of Lemma 4.4:

**Lemma 4.11** *If any processor reads a counter value  $v \geq 2n$ , then all subsequent reads will return values  $\geq n+1$ ; in the symmetric case where  $v \leq -2n$ , all subsequent reads will return values  $\leq -(n+1)$ .*

**Proof:** Apply the proof of Lemma 4.4 with  $K = n$ . ■

Similarly, the proof that the counter  $c$  does not overflow is a straightforward modification of Lemma 4.5:

**Lemma 4.12** *The value of  $c$  never leaves the range  $[-4n, 4n]$  in any execution of the consensus protocol.*

**Proof:** Apply the proof of Lemma 4.5 with  $K = n$ . ■

Termination is trickier to demonstrate. As in the case of the shared coin, the key to proving the consensus protocol's termination is the fact that the scheduler's only alternative to moving the true position randomly is to move it away from the origin. In the shared coin protocol, this condition depends on fixing the parameter  $K \geq n$ . In the consensus protocol the situation is more complicated, as the protocol uses its knowledge of the number of currently active processors to set the inner boundaries of the slope close to the origin while still preventing the scheduler from being able to force the true position to move toward the origin.

**Lemma 4.13** *Let  $n$  be the total number of processors and  $p$  be the number of processors that take at least one step before some processor decides on a value. Then the worst-case expected running time of the consensus protocol is  $O(p^2 + n)$  total counter operations.*

**Proof:** We will show that the consensus protocol terminates in  $O(p^2 + n)$  time by reducing it to a controlled random walk of the true position  $t$ . Divide the execution of the protocol into two phases. In the first phase, at most one of  $a_0, a_1$  is nonzero; if the execution does not leave the first phase before

$2n$  increments or decrements have occurred the protocol will terminate after  $O(n)$  additional steps by Lemma 4.11.

In the second phase, both  $a_0$  and  $a_1$  are nonzero. Let  $v$  be a value read by some processor from the counter  $c$ . By Lemma 4.10 we know that  $|t - v| \leq a_0 + a_1 - 1 \leq p - 1$ . Now, to force an increment during the second phase the scheduler must show a processor a counter value  $v$  that is at least  $a_0 + a_1$ , possibly by withholding local coin-flips to raise the value of  $c$  or by withholding increments to lower the value of  $a_0 + a_1$ . In either case Lemma 4.10 applies and  $t$  must be greater than 0. The case of the scheduler attempting to force a decrement is symmetric, and thus in either case the scheduler can only force the true position to move away from 0.

Furthermore, since  $p$  is an upper bound both on the distance between  $c$  and  $t$  and on the value of  $a_0 + a_1$ , if  $|t| \geq 2p$  then  $|v| \geq a_0 + a_1$  and the true position will move away from 0 thereafter. Thus the second phase of the execution can be modeled as a controlled random walk in the sense of Lemma 4.3 with center 0, barriers at  $\pm 2p$ , and a starting position equal to the true position at the end of the first phase. By Lemma 4.3, this random walk will take an expected  $O(p^2)$  steps, each consisting of a constant number of counter operations; to this value must be added  $O(n)$  steps until termination, up to  $O(n)$  steps from the first phase, and  $O(p^2)$  extra read operations due to extra passes through the loop in `scan_counters()`. The total expected number of counter operations is thus  $O(p^2 + n)$ . ■

Note that the expected running time of  $O(p^2 + n)$  is expressed in total *counter* operations. If the counter is implemented as described in Section 4.3 the total number of *register* operations will be  $O(n^2(p^2 + n))$ .

**Lemma 4.14** *The protocol of Figure 4.5 satisfies the validity condition.*

**Proof:** Suppose every processor starts with the input 1. Then  $a_0$  is never incremented and so retains its initial value of 0 throughout the execution of the protocol. Thus each processor will increment  $c$  until it reads a value  $v \geq 2n$  at which point it will decide 1. The case where every processor has input 0 is symmetric. ■

Combining the lemmas gives:

**Theorem 4.15** *Figure 4.5 implements a consensus protocol.*

**Proof:** Lemmas 4.11, 4.13, and 4.14. ■

It is worth looking at the behavior of the shared coin implicitly embedded in the consensus protocol of Figure 4.5. Because the function of detecting agreement is implemented in the shared coin itself, limiting scheduler control over the outcome of the shared coin is no longer necessary to achieve consensus. Thus the parameter  $K$  of the shared coin protocol can be set to minimize the time taken in the random walk without regard to its effect on the agreement parameter  $\delta$ . In the protocol of Figure 4.5 the shared coin has an effective agreement parameter of  $\frac{1}{2p}$ , as low as is possible without setting  $K < p$ .

At the same time, the simplicity of the protocol allows the number and size of the shared counters to be very small. Unfortunately, when the available primitives are limited to atomic registers this small size is lost in the  $\Theta(n^2)$  space overhead of the atomic scan operation. It is not immediately clear that this overhead is a necessary feature of an atomic counter implementation; much work remains to be done in this area.

# Chapter 5

## Consensus Using Weighted Voting

### 5.1 Introduction

In the previous chapter we built a consensus protocol that directly incorporated a robust shared coin. Here we will show how to construct a faster but non-robust shared coin which gives consensus using standard constructions such as the one of [AH90a] described in Section 3.3.

This shared coin protocol requires a departure from previous practice. As in the protocols of the previous chapter, the fundamental technique behind all shared coin protocols since [AH90a] has been the use of repeated, equally-weighted votes to reduce the impact of any particular processor's private knowledge and with it the adversary's ability to affect the outcome of the coin. There are many advantages to this approach. The processors act as anonymous conduits of a stream of unpredictable random increments. If the scheduler stops a particular processor, at worst all it does is keep one vote from being written out to the common pool—the next local coin flip executed by some other processor is no more or less likely to give the value the scheduler wants than the next one executed by the processor it has just stopped. Intuitively, the scheduler's power over the outcome of the shared coin is limited to filtering out up to  $n - 1$  local coin flips from this stream of independent random variables. But the effect of this filtering is at worst equivalent to adjusting the final tally of votes by up to  $n - 1$ . If a constant

multiple of  $n^2$  votes are cast, the total variance will be  $\Omega(n^2)$ . Because the total vote is approximately normally distributed, the protocol can guarantee that with constant probability the total vote is more than  $n$  away from the origin, rendering the scheduler's adjustment ineffective.

Alas, the very anonymity of the processors that is the strength of the voting technique is also its greatest weakness. To overcome the scheduler's power to withhold votes, it is necessary that a total of  $\Omega(n^2)$  votes are cast—but the scheduler might also choose to stop all but one of the processors, leaving that lone processor to generate all  $\Omega(n^2)$  votes by itself. It follows that, for all of the polynomial-time wait-free consensus protocols based on unweighted voting, the worst-case expected bound on the work done by a single processor is asymptotically no better than the bound on the total work done by all of the processors together.

In this chapter we show how to avoid this problem by modifying a protocol of Bracha and Rachman [BR91] to allow the processor to cast votes of increasing weight. Thus a fast processor or a processor running in isolation can quickly generate votes of sufficient total variance to finish the protocol, at the cost of giving the scheduler greater control by allowing it both to withhold votes with larger impact and to choose among up to  $n$  different weights (one for each processor) when determining the weight of the next vote.

There are two main difficulties that this approach entails. The first is that careful adjustment of the weight function and other parameters of the protocol is necessary to make sure that it performs correctly. More importantly, allowing the weight of the  $i$ -th vote to depend on the particular processor the scheduler chooses to run, which may in turn depend on the outcomes of previous votes, means that we cannot treat the sequence of votes as a sequence of independent random variables.

However, the *sign* of each vote is determined by a fair coin flip that the scheduler cannot predict in advance, and so despite all the scheduler's powers, the expected value of each vote before it is cast is always 0. This is the primary requirement of a **martingale process** [Bil86, Fel71, Kop84]. Under the right conditions, martingales have many similarities to sequences of sums of independent random variables. In particular, martingale analogues of the Central Limit Theorem and Chernoff bounds will be used in the proof of correctness.

The rest of the chapter is organized as follows. Section 5.2 defines the shared coin protocol and gives an overview of its operation. Section 5.3

```

1 procedure shared_coin()
2 begin
3   my_reg(variance, vote) ← (0, 0)
4   t ← 1
5   repeat
6     for i = 1 to c do
7       vote ← local_flip() × w(i)
8       my_reg ← (my_reg.variance + w(t)2, my_reg.vote + vote)
9       t ← t + 1
10    end
11    read all the registers, summing the variance fields into the
12      local variable total_variance
13    until total_variance > k
14    read all the registers, summing the vote fields into the local vari-
15      able total_vote
16    if total_vote > 0
17      then output 1
18    else if total_vote < 0
19      then output 0
20    else fail
21 end

```

Figure 5.1: Shared coin protocol.

contains a brief definition of martingales and describes some of their properties. Finally, Section 5.4 proves the correctness of the protocol for two sets of parameters, one of which allows it to simulate the equally-weighted voting protocol of [BR91], and one which gives a bound of  $O(n \log^2 n)$  on the expected number of operations executed by a single processor.

## 5.2 The shared coin protocol

Figure 5.1 gives pseudocode for each processor's behavior during the shared coin protocol. Each processor repeatedly flips a local coin that returns the values  $+1$  and  $-1$  with equal probability. The weighted value of

each flip is  $w(t)$  or  $-w(t)$  respectively, where  $t$  is the number of coins flipped by the processor up to and including its current flip. Each weighted flip represents a vote for either the output value 1 (if positive) or 0 (if negative). After each flip, the processor updates its register to hold the sum of the weighted flips it has performed, and the sum of the squares of their values. After every  $c$  flips, the processor reads the registers of all the other processors, and computes the sum of all the weighted flips (the total vote) and the sum of the squares of their values (the total variance). If the total variance is greater than the quorum  $K$ , it stops, and outputs 1 if the total vote is positive, and 0 if it is negative (it treats a total vote of zero as a failure to avoid introducing asymmetry between the two outcomes). Alternatively, if the total variance has not yet reached the quorum  $K$ , it continues to flip its local coin.

As in the previous chapter, the function *local\_flip* returns the values 1 and -1 randomly with equal probability. The values  $K$  and  $c$  are parameters of the protocol which will be set depending on the number of processors  $n$  to give the desired bounds on the agreement parameter and running time. The weight function  $w(t)$  is used to make later local coin flips have more effect than earlier ones; so that a processor running in isolation will be able to achieve the quorum  $K$  quickly. The weight function will be assumed to be of the form  $w(t) = t^a$  where  $a$  is a nonnegative parameter depending on  $n$ ; though other weight functions are possible, this choice simplifies the analysis.

We will demonstrate that for suitable choice of  $K$ ,  $c$  and  $a$  all processors return 1 with constant probability; the case of all processors returning 0 will follow by symmetry. The structure of the argument follows the proof of correctness of the less sophisticated protocol of Bracha and Rachman [BR91], which corresponds to Figure 5.1 when  $w(t)$  is the constant 1,  $K = \Theta(n^2)$ , and  $c = \Theta(n/\log n)$ . Votes cast before the quorum  $K$  is reached will form a pool of **common votes** that all processors see.<sup>1</sup> We will show that with constant probability (i) the total of the common votes is far from the origin and (ii) the sum of the **extra votes** cast between the time the quorum is reached and the time some processor does its final read in line 13 is small, so that the total vote read by each processor will have the same sign as the total common vote.

---

<sup>1</sup>The definitions of the common and extra votes we will use differ slightly from those used in [BR91]; the formal definitions appear in Section 5.4.

This simple overview of the proof hides many tricky details. To simplify the analysis we will concentrate not on the votes actually written to the registers but on the votes whose values have been **decided** by the processors' execution of the local coin flip in line 7; conversion back to the values actually in the registers will be done by showing a bound on the difference between the total decided vote and the total of the register values. In effect, we are treating a vote as having been "cast" the moment that its value is determined, instead of when it becomes visible to the other processors.

Some care is also needed to correctly model the sequence of votes. Most importantly, as pointed out above, allowing the weight of the  $i$ -th vote to depend on which processor the scheduler chooses to run means the votes are not independent. So the straightforward proof techniques used for protocols based on a stream of identically-distributed random votes no longer apply, and it is necessary to bring in the theory of martingales to describe the execution of the protocol.

### 5.3 Martingales

A **martingale** is a sequence of random variables  $S_1, S_2, \dots$ , which informally may be thought of as representing the changes in the fortune of a gambler playing in a fair casino. Because the gambler can choose how much to bet or which game to play at each instant, each random variable  $S_i$  may depend on all previous events. But because the casino is fair and the gambler cannot predict the future, the expected change in the gambler's fortune at any play is always 0.

We will need to use a very general definition of a martingale [Bil86, Fel71, Kop84]. The simplest definition of a martingale says that the expected value of  $S_{i+1}$  given  $S_1, S_2, \dots, S_i$  is just  $S_i$ . To use a gambling analogy, this definition says that a gambler who knows only the previous values of her fortune cannot predict its expected future value any better than by simply using its current value. But what if the gambler knows more information than just the changing size of her bankroll? For example, imagine that she is placing bets on a fair version of roulette, and always bets on either red or black. Knowing that her fortune increased after betting red will tell her only that one of eighteen red numbers came up; but a real gambler will see precisely *which* of the eighteen numbers it was. Still, we would like to claim that this

additional knowledge does not affect her ability to predict the future. To do so, the definition of a martingale must be extended to allow additional information to be represented explicitly.

The tool used to represent the information known at any point in time will be a concept from measure theory, a  $\sigma$ -algebra<sup>2</sup> The description given here is informal; more complete definitions can be found in [Fel71, Sections IV.3, IV.4, and V.11] or [Bil86].

### 5.3.1 Knowledge, $\sigma$ -algebras, and measurability

Recall that any probabilistic statement is always made in the context of some (possibly implicit) **sample space**. The elements of the sample space (called **sample points**) represent all possible results of some set of experiments, such as flipping a sequence of coins or choosing a point at random from the unit interval. Intuitively, all randomness is reduced to selecting a single point from the sample space. An **event**, such as a particular coin-flip coming up heads or a random variable taking on the value 0, is simply a subset of the sample space that “occurs” if one of the sample points it contains is selected.

If we are omniscient, we can see which sample point is chosen and thus can tell for each event whether it occurs or not. However, if we have only partial information, we will not be able to determine whether some events occurred or not. We can represent the extent of our knowledge by making a list of all events we do know about. This list will have to satisfy certain closure properties; for example, if we know whether or not  $A$  occurred, and whether or not  $B$  occurred, then we should know whether or not the event “ $A$  or  $B$ ” occurred.

We will require that the set of known events be a  $\sigma$ -algebra. A  $\sigma$ -algebra  $\mathcal{F}$  is a family of subsets of a sample space  $\Omega$  that (i) contains the empty set; (ii) is closed under complement: if  $\mathcal{F}$  contains  $A$ , it contains  $\Omega \setminus A$  (the complement of  $A$ ); and (iii) is closed under countable union: if  $\mathcal{F}$  contains all of  $A_1, A_2, \dots$ , it contains  $\bigcup_{i=1}^{\infty} A_i$ .<sup>3</sup> An event  $A$  is said to be  $\mathcal{F}$ -measurable if it is contained in  $\mathcal{F}$ . In our context, the term “measurable,” which comes from the original measure-theoretic use of  $\sigma$ -algebras to represent families of sets on which a probability distribution is well-defined, simply means “known.”

---

<sup>2</sup>Sometimes called a  $\sigma$ -field.

<sup>3</sup>Additional properties, such as being closed under finite union or intersection, follow immediately from this definition.

We “know” about an event if we can determine whether or not it occurred. What about random variables? A random variable  $X$  is defined to be  $\mathcal{F}$ -measurable if every event of the form  $X \leq c$  is  $\mathcal{F}$ -measurable. (The closure properties of  $\mathcal{F}$  then imply that such events as  $a \leq X < b$ ,  $X = d$ , and so forth are also  $\mathcal{F}$ -measurable.) Looking at the situation in reverse, given random variables  $X_1, X_2, \dots$  we can consider the minimum  $\sigma$ -algebra  $\mathcal{F}$  for which each of the random variables is  $\mathcal{F}$ -measurable; this  $\sigma$ -algebra, written  $\langle X_i \rangle$ , is called the  $\sigma$ -algebra generated by  $X_1, X_2, \dots$ , and represents all information that can be inferred from knowing the values of the generators.

A  $\sigma$ -algebra gives us a rigorous way to define “knowledge” in a probabilistic context. Measurability and generated  $\sigma$ -algebras give us a way to move back and forth between the abstract concept of a  $\sigma$ -algebra and concrete statements about which random variables are completely known. To analyze random variables that are only *partially* known, we need one more definition. We need to extend conditional expectations so that the condition can be a  $\sigma$ -algebra rather than just a collection of random variables.

For each event  $A$  let  $I_A$  be the indicator variable that is 1 if  $A$  occurs and 0 otherwise. Let  $U = E[X | \mathcal{F}]$  be a random variable such that (i)  $U$  is  $\mathcal{F}$ -measurable and (ii)  $E[U I_A] = E[X I_A]$  for all  $A$  in  $\mathcal{F}$ . The random variable  $E[X | \mathcal{F}]$  is called the **conditional expectation** of  $X$  with respect to  $\mathcal{F}$  [Fel71, Section V.11]. Intuitively, the first condition on  $E[X | \mathcal{F}]$  says that it reveals no information not already found in  $\mathcal{F}$ . The second condition says that just knowing that some event in  $\mathcal{F}$  occurred does not allow one to distinguish between  $X$  and  $E[X | \mathcal{F}]$ ; this fact ultimately implies that  $E[X | \mathcal{F}]$  uses all information that is found in  $\mathcal{F}$  and is relevant to  $X$ .

If  $\mathcal{F}$  is generated by random variables  $X_1, X_2, \dots$ , the conditional expectation  $E[X | \mathcal{F}]$  reduces to the simpler version  $E[X | X_1, X_2, \dots]$ . Some other facts about conditional expectation that we will use (but not prove): if  $X$  is  $\mathcal{F}$ -measurable, then  $E[XY | \mathcal{F}] = X E[Y | \mathcal{F}]$  (which implies  $E[X | \mathcal{F}] = X$ ); and if  $\mathcal{F}' \subseteq \mathcal{F}$ , then  $E[E[X | \mathcal{F}] | \mathcal{F}'] = E[X | \mathcal{F}']$ . See [Fel71, Section V.11].

### 5.3.2 Definition of a martingale

We now have the tools to define a martingale when the information available at each point in time is not limited to just the values of earlier random variables. A **martingale**  $\{S_i, \mathcal{F}_i\}, 1 \leq i \leq n$ , is a stochastic process where each  $S_i$  is a random variable representing the state of the process at time  $i$  and

$\mathcal{F}_i$  is a  $\sigma$ -algebra representing the knowledge of the underlying probability distribution available at time  $i$ . Martingales are required to satisfy three axioms, for all  $i$ :

1.  $\mathcal{F}_i \subseteq \mathcal{F}_{i+1}$ . (The past is never forgotten.)
2.  $S_i$  is  $\mathcal{F}_i$ -measurable. (The present is always known.)
3.  $E[S_{i+1} | \mathcal{F}_i] = S_i$ . (The future cannot be foreseen.)

Often  $\mathcal{F}_i$  will simply be the  $\sigma$ -algebra  $\langle S_1, \dots, S_i \rangle$  generated by the variables  $S_1$  through  $S_i$ ; in this case axioms 1 and 2 will hold automatically.

To avoid special cases let  $\mathcal{F}_0$  denote the trivial  $\sigma$ -algebra consisting of the empty set and the entire probability space. The **difference sequence** of a martingale is the sequence  $X_1, X_2, \dots, X_n$  where  $X_1 = S_1$  and  $X_i = S_i - S_{i-1}$  for  $i > 1$ . A **zero-mean martingale** is a martingale for which  $E[S_i] = 0$ .

### 5.3.3 Gambling systems

A remarkably useful theorem, which has its origins in the study of gambling systems, is due to Halmos [Hal39]. We restate his theorem below in modern notation:

**Theorem 5.1** *Let  $\{S_i, \mathcal{F}_i\}, 1 \leq i \leq n$  be a martingale with difference sequence  $\{X_i\}$ . Let  $\{\zeta_i\}, 1 \leq i \leq n$  be random variables taking on the values 0 and 1, such that each  $\zeta_i$  is  $\mathcal{F}_{i-1}$ -measurable. Then the sequence of random variables  $S'_i = \sum_{j=1}^i \zeta_j X_j$  is a martingale relative to  $\mathcal{F}_i$ .*

**Proof:** The first two properties are easily verified. Because  $\zeta_i$  is  $\mathcal{F}_{i-1}$ -measurable,  $E[\zeta_i X_i | \mathcal{F}_{i-1}] = \zeta_i E[X_i | \mathcal{F}_{i-1}] = 0$ , and the third property also follows. ■

### 5.3.4 Limit theorems

Many results that hold for sums of independent random variables carry over in modified form to martingales. For example, the following theorem of Hall and Heyde [HH80, Theorem 3.9] is a martingale version of the classical Central Limit Theorem:

**Theorem 5.2 ([HH80])** Let  $\{S_i, \mathcal{F}_i\}$  be a zero-mean martingale. Let  $V_n^2 = \sum_{i=1}^n \mathbb{E}[X_i^2 | \mathcal{F}_{i-1}]$  and let  $0 < \delta \leq 1$ . Define  $L_n = \sum_{i=1}^n \mathbb{E}[|X_i|^{2+2\delta}] + \mathbb{E}[|V_n^2 - 1|^{1+\delta}]$ . Then there exists a constant  $C$  depending only on  $\delta$  such that whenever  $L_n \leq 1$ ,

$$|\Pr[S_n \leq x] - \Phi(x)| \leq C L_n^{1/(3+2\delta)} \left[ \frac{1}{1 + |x|^{4(1+\delta)^2/(3+2\delta)}} \right], \quad (5.1)$$

where  $\Phi$  is the standard unit normal distribution with mean 0 and variance 1.

If we are interested only in the tails of the distribution of  $S_n$ , we can get a tighter bound using Azuma's inequality, a martingale analogue of the standard Chernoff bound [Che52] for sums of independent random variables. The usual form of this bound (see [AS92, Spe87]) assumes that the difference variables  $X_i$  satisfy  $|X_i| \leq 1$ . This restriction is too severe for our purposes, so below we prove a generalization of the inequality. In order to do so we will need the following technical lemma.

**Lemma 5.3** Let  $\{S_i, \mathcal{F}_i\}, 1 \leq i \leq n$  be a zero-mean martingale with difference sequence  $\{X_i\}$ . Let  $\mathcal{F}_0 \subseteq \mathcal{F}_1$  be a (not necessarily trivial)  $\sigma$ -algebra such that  $\mathbb{E}[S_1 | \mathcal{F}_0] = 0$ . If there exists a sequence of random variables  $w_1, w_2, \dots, w_n$ , and a random variable  $W$ , such that

1.  $W$  is  $\mathcal{F}_0$ -measurable,
2. Each  $w_i$  is  $\mathcal{F}_{i-1}$ -measurable,
3. For all  $i$ ,  $|X_i| \leq w_i$  with probability 1, and
4.  $\sum_{i=1}^n w_i^2 \leq W$  with probability 1,

then for any  $\alpha > 0$ ,

$$\mathbb{E}[e^{\alpha S_n} | \mathcal{F}_0] \leq e^{\alpha^2 W/2} \quad (5.2)$$

**Proof:** The proof is by induction on  $n$ . Using the convexity of  $e^{\alpha x}$  and the fact that  $\mathbb{E}[X_1 | \mathcal{F}_0] = 0$ , we have

$$\mathbb{E}[e^{\alpha X_1} | \mathcal{F}_0] \leq \frac{1}{2} (e^{-\alpha w_1} + e^{\alpha w_1}) = \cosh \alpha w_1 \leq e^{\alpha^2 w_1^2/2}.$$

If  $n = 1$  we are done, since  $w_1^2 \leq W$ . If  $n$  is greater than 1, for each  $i \leq n-1$  let  $S'_i = S_{i+1} - X_1$  and  $\mathcal{F}'_i = \mathcal{F}_{i+1}$ . Then  $\{S'_i, \mathcal{F}'_i\}, 1 \leq i \leq n-1$  satisfies the conditions of the lemma with  $\mathcal{F}'_0 = \mathcal{F}_1$ ,  $w'_i = w_{i+1}$  and  $W' = W - w_1^2$ , so by the induction hypothesis  $E[e^{\alpha S'_{n-1}} | \mathcal{F}'_0] \leq e^{\alpha^2(W-w_1^2)/2}$ . But then, using the fact that  $E[X | \mathcal{F}] = E[E[X | \mathcal{F}'] | \mathcal{F}]$  when  $\mathcal{F} \subseteq \mathcal{F}'$ , we can compute:

$$\begin{aligned}
 E[e^{\alpha S_n} | \mathcal{F}_0] &= E\left[E\left[e^{\alpha X_1} e^{\alpha(S_n - X_1)} | \mathcal{F}_1\right] | \mathcal{F}_0\right] \\
 &= E\left[e^{\alpha X_1} E\left[e^{\alpha S'_{n-1}} | \mathcal{F}'_0\right] | \mathcal{F}_0\right] \\
 &\leq E\left[e^{\alpha X_1} e^{\alpha^2(W-w_1^2)/2} | \mathcal{F}_0\right] \\
 &= e^{\alpha^2(W-w_1^2)/2} E\left[e^{\alpha X_1} | \mathcal{F}_0\right] \\
 &\leq e^{\alpha^2(W-w_1^2)/2} e^{\alpha^2 w_1^2/2} \\
 &= e^{\alpha^2 W/2}.
 \end{aligned}$$

■

**Theorem 5.4** Let  $\{S_i, \mathcal{F}_i\}, 1 \leq i \leq n$  be a zero-mean martingale with difference sequence  $\{X_i\}$ . If there exists a sequence of random variables  $w_1, w_2, \dots, w_n$ , and a constant  $W$ , such that

1. Each  $w_i$  is  $\mathcal{F}_{i-1}$ -measurable.
2. For all  $i$ ,  $|X_i| \leq w_i$  with probability 1, and
3.  $\sum_{i=1}^n w_i^2 \leq W$  with probability 1,

then for any  $\lambda > 0$ ,

$$\Pr[S_n \geq \lambda] \leq e^{-\lambda^2/2W}. \quad (5.3)$$

**Proof:** By Lemma 5.3, for any  $\alpha > 0$ ,  $E[e^{\alpha S_n}] \leq e^{\alpha^2 W/2}$ . Thus by Markov's inequality

$$\Pr[S_n \geq \lambda] = \Pr\left[e^{\alpha S_n} \geq e^{\alpha \lambda}\right] \leq e^{\alpha^2 W/2} e^{-\alpha \lambda}.$$

Setting  $\alpha = \lambda/W$  gives (5.3). ■

Symmetry immediately gives us:

**Corollary 5.5** For any martingale  $\{S_i, \mathcal{F}_i\}$  satisfying the premises of Theorem 5.4, and any  $\lambda > 0$

$$\Pr[S_n \leq -\lambda] \leq e^{-\lambda^2/2W}. \quad (5.4)$$

**Proof:** Replace each  $S_i$  by  $-S_i$  and apply Theorem 5.4. ■

## 5.4 Proof of correctness

For this section we will fix a particular scheduler. We may assume without loss of generality that the scheduler is deterministic, because any random inputs the scheduler might use cannot depend on the history of an execution and therefore may also be fixed in advance.

Consider the sequence of random variables  $X_1, X_2, \dots$  where  $X_i$  represents the  $i$ -th vote that is decided by some processor executing line 7, or 0 if fewer than  $i$  local coin flips occur. For each  $i$  let  $\mathcal{F}_i$  be  $\langle X_1 \dots X_i \rangle$ , the  $\sigma$ -algebra generated by  $X_1$  through  $X_i$ . Because the scheduler is deterministic, all of the random events in the system preceding the  $i$ -th vote are captured in the variables  $X_1$  through  $X_{i-1}$ , and the  $\sigma$ -algebra  $\mathcal{F}_{i-1}$  thus determines the entire history of the system up to but not including the  $i$ -th vote. Furthermore, since the scheduler's behavior depends only on the history of the system,  $\mathcal{F}_{i-1}$  in fact determines the scheduler's choice of which processor will cast the  $i$ -th vote. Thus conditioned on  $\mathcal{F}_{i-1}$ ,  $X_i$  is just a random variable which takes on the values  $\pm w$  with equal probability for some weight  $w$  determined by the scheduler's choice of which processor to run. Hence  $E[X_i | \mathcal{F}_{i-1}] = 0$ , and the sequence of partial sums  $S_i = \sum_{j=1}^i X_j$  is a martingale relative to  $\{\mathcal{F}_i\}$ .

We are not going to analyze  $\{S_i, \mathcal{F}_i\}$  directly. Instead, it will be used as a base on which other martingales will be built using Theorem 5.1.

Let  $\kappa_i = 1$  if  $\sum_{j=1}^i X_j^2 \leq K$  and 0 otherwise. Votes for which  $\kappa_i = 1$  will be called **common votes**. For each processor  $P$  let  $\zeta_{P,i} = 1$  if the vote  $X_i$  occurs before  $P$  reads, during its final read in line 13, the register of the processor deciding  $X_i$ , and let  $\zeta_{P,i} = 0$  otherwise. In effect,  $\zeta_{P,i}$  is the indicator variable for whether  $P$  would see  $X_i$  if it were written out immediately. Observe that for a fixed scheduler the values of both  $\kappa_i$  and  $\zeta_{P,i}$  can be determined by examining the history of the system up to but not including the time when the

vote  $X_i$  is cast, and thus both  $\kappa_i$  and  $\zeta_{P,i}$  are  $\mathcal{F}_{i-1}$ -measurable. Consequently the sequences  $\{\sum_{j=1}^i \kappa_j X_j\}$  and  $\{\sum_{j=1}^i \zeta_{P,j} X_j\}$  are martingales relative to  $\{\mathcal{F}_i\}$  by Theorem 5.1. Votes for which  $\zeta_{P,i} = 1$  but  $\kappa_i = 0$  will be referred to as the **extra votes** for processor  $P$ . (Observe that  $\zeta_{P,i} \geq \kappa_i$  since  $P$  could not have started its final read until the total variance was at least  $K$ .) The sequence  $\{\sum_{j=1}^i (\zeta_{P,j} - \kappa_j) X_j\}$  of the partial sums of these extra votes is a difference of martingales and is thus also a martingale relative to  $\{\mathcal{F}_i\}$ .

The structure of the proof of correctness is as follows. First, we show that the distribution of the total common vote,  $\sum \kappa_i X_i$ , is close to a normal distribution with mean 0 and variance  $K$  for suitable choices of  $a$  and  $K$ ; in particular, for  $n$  sufficiently large, the probability that  $\sum \kappa_i X_i > x\sqrt{K}$  will be at least a constant for any fixed  $x$ . Next, we complete the proof by showing that if the total common vote is far from the origin the chances that any processor will read a total vote whose sign differs from the common vote is small. This fact is itself shown in two steps. First, it is shown that, for suitable choice of  $c$ , the total of the extra votes for a processor  $P$ ,  $\sum (\zeta_{P,i} - \kappa_i) X_i$ , will be small with high probability. Second, a bound  $\Delta$  is derived on the difference between  $\sum \zeta_{P,i} X_i$  and the total vote actually read by  $P$ .

It will be necessary to select values for  $a$ ,  $K$ , and  $c$  that give the correct bounds on the probabilities. However, we will be in a better position to justify our choice for these parameters after we have developed more of the analysis, so the choice of parameters will be deferred until Section 5.4.5.

#### 5.4.1 Phases of the protocol

We begin by defining the phases of the protocol more carefully. Let  $t_i$  be the value of the  $i$ -th processor's internal variable  $t$  at any given step of the protocol. Let  $U_i$  be the random variable representing the maximum value of  $t_i$  during the entire execution of the protocol. Let  $T_i$  be the random variable representing the maximum value of  $t_i$  during the part of the execution of the protocol where  $\kappa_i = 1$ .

In the proof of correctness we will encounter many quantities of the form  $\sum_{i=1}^n \xi(T_i)$  or  $\sum_{i=1}^n \xi(U_i)$  for various functions  $\xi$ . We will want to get bounds on these quantities without having to look too closely at the particular values of each  $T_i$  or  $U_i$ . This section proves several very general inequalities about

quantities of this form, all of which are ultimately based on the following constraint:

$$K \geq \sum_i \sum_{j=1}^{T_i} j^{2a} \geq \sum_i \int_0^{T_i} j^{2a} dj = \sum_i \frac{T_i^{2a+1}}{2a+1}. \quad (5.5)$$

The constant  $2a + 1$  will reappear often; for convenience we will write it as  $A$ . As noted above,  $a \geq 0$ , and hence  $A \geq 1$ .

Define  $T_K = \left(\frac{AK}{n}\right)^{1/A}$ , so that  $K = \frac{nT_K^A}{A}$ . The constant  $T_K$  represents the maximum value of each  $T_i$  if they are set to be equal while satisfying inequality (5.5). Note that  $T_K$  need not be integral. Now we can show:

**Lemma 5.6** *Let  $\psi(x) = x^A/A$  and let  $\chi$  be any strictly increasing function such that  $\chi\psi^{-1}$  is concave. Then for any non-negative  $\{x_i\}$ , if  $\sum_{i=1}^n \psi(x_i) \leq K$ , then  $\sum_{i=1}^n \chi(x_i) \leq n\chi(T_K)$ .*

**Proof:** Since  $\chi\psi^{-1}$  is concave, we have

$$\chi^{-1} \left( \sum \frac{\chi(x_i)}{n} \right) \leq \psi^{-1} \left( \sum \frac{\psi(x_i)}{n} \right)$$

[HLP52, Theorem 92]. Simple algebraic manipulation yields

$$\sum \chi(x_i) \leq n\chi \left( \psi^{-1} \left( \sum \frac{\psi(x_i)}{n} \right) \right)$$

But

$$\psi^{-1} \left( \sum \frac{\psi(x_i)}{n} \right) = \psi^{-1} \left( \frac{1}{n} \sum \frac{x_i^A}{A} \right) \leq \psi^{-1} \left( \frac{K}{n} \right) = T_K.$$

Hence  $\sum \chi(x_i) \leq n\chi(T_K)$ . ■

Letting  $\chi$  be the identity function we have  $\chi\psi^{-1}(x) = (Ax)^{1/A}$ , which is concave for  $A \geq 1$ . Hence:

**Corollary 5.7**

$$\sum_{i=1}^n T_i \leq nT_K. \quad (5.6)$$

In the case where  $\chi\psi^{-1}$  is convex, the following lemma applies instead:

**Lemma 5.8** Let  $\psi(x) = x^A/A$  and let  $\chi$  be any strictly increasing function such that  $\chi\psi^{-1}$  is convex. Then for any non-negative  $\{x_i\}$ , if  $\sum_{i=1}^n x_i^A/A \leq K$ , then  $\sum_{i=1}^n \chi(x_i) \leq (n-1)\chi(0) + \chi(n^{1/A}T_K)$ .

**Proof:** Let  $Y = \sum \psi(x_i)$ . Now  $\chi(x_i) = \chi\psi^{-1}\psi(x_i)$  or

$$\chi\psi^{-1} \left( \left( 1 - \frac{\psi(x_i)}{Y} \right) 0 + \frac{\psi(x_i)}{Y} Y \right)$$

which is at most

$$\left( 1 - \frac{\psi(x_i)}{Y} \right) \chi\psi^{-1}(0) + \frac{\psi(x_i)}{Y} \chi\psi^{-1}(Y)$$

given the convexity of  $\chi\psi^{-1}$ . Hence

$$\begin{aligned} \sum_{i=1}^n \chi(x_i) &\leq n\chi\psi^{-1}(0) - \left( \sum_{i=1}^n \frac{\psi(x_i)}{Y} \right) \chi\psi^{-1}(0) + \left( \sum_{i=1}^n \frac{\psi(x_i)}{Y} \right) \chi\psi^{-1}(Y) \\ &= (n-1)\chi\psi^{-1}(0) + \chi\psi^{-1} \left( \sum_{i=1}^n \psi(x_i) \right) \\ &\leq (n-1)\chi\psi^{-1}(0) + \chi\psi^{-1}(K) \end{aligned}$$

which is just  $(n-1)\chi(0) + \chi(n^{1/A}T_K)$ . ■

The quantity  $n^{1/A}T_K$  is the maximum value that any  $x_i$  can take on without violating the constraint on  $\sum x_i$ . So what Lemma 5.8 says is that if  $\chi\psi^{-1}$  is convex,  $\sum \chi(x_i)$  is maximized by maximizing one of the  $x_i$  while setting the rest to zero.

For the variables  $U_i$  we can show:

**Lemma 5.9** Let  $\psi(x) = x^A/A$  and let  $\chi$  be any strictly increasing function such that  $\chi(\psi^{-1}(x) + c + 1)$  is concave in  $x$ . Then for any non-negative  $\{x_i\}$ , if  $\sum_{i=1}^n \psi(x_i) \leq K$ , then

$$\sum_{i=1}^n \chi(U_i) \leq n\chi(T_K + c + 1) \quad (5.7)$$

**Proof:** Let  $W_i$  be the number of votes *written* to the registers during the part of the execution where the total of the register variance fields is less than or equal to  $K$ . The set of variables  $\{W_i\}$  satisfies the inequality  $\sum W_i^A/A \leq K$  using the same argument as gives (5.5). Furthermore  $U_i \leq W_i + 1 + c$ , because after the  $i$ -th processor's next vote the total variance in the registers must exceed  $K$  and it can cast at most  $c$  more votes before noticing this fact. Define  $\chi'(x) = \chi(x + c + 1)$ . Then  $\chi(U_i) \leq \chi(W_i + c + 1) = \chi'(W_i)$ . But  $\psi, \chi'$  satisfy the premises of Lemma 5.6 and thus  $\sum_{i=1}^n \chi(U_i) \leq \sum_{i=1}^n \chi'(W_i) \leq n\chi'(T_K) = n\chi(T_K + c + 1)$ . ■

Setting  $\chi(x)$  to  $x$  gives

**Corollary 5.10**

$$\sum_{i=1}^n U_i \leq n(T_K + c + 1) \quad (5.8)$$

**Proof:**  $\chi(\psi^{-1}(x) + c + 1) = Ax^{1/A} + c + 1$ , which is concave since  $A \geq 1$ . ■

Define  $g = 1 + \frac{c+3}{T_K}$ ; then  $gT_K = T_K + c + 3$  will be an upper bound for  $T_K + c + 1$  as well as a number of closely related constants involving  $c$  that will appear later.

### 5.4.2 Common votes

The purpose of this section is to show that for  $n$  sufficiently large, the total common vote is far from the origin with constant probability. We do so by showing that under the right conditions the total common vote will be nearly normally distributed.

Let  $S_{K,i} = \sum_{j=1}^i \kappa_j X_j$ . As pointed out above,  $\{S_{K,i} = \sum_{j=1}^i \kappa_j X_j, \mathcal{F}_i\}$  is a martingale. Let  $N = \lceil nT_K \rceil$ . It follows from Corollary 5.7 that  $\kappa_i = 0$  for  $i > N$  and thus  $S_{K,N} = \lim_{i \rightarrow \infty} S_{K,i}$  is the sum of all the common votes. The distribution of  $S_{K,N}$  is characterized in the following lemma.

**Lemma 5.11** *If*

$$\frac{6A^2}{n^{1/A} T_K} \leq 1, \quad (5.9)$$

*then for any  $x$ ,*

$$|\Pr[S_{K,N} \leq x\sqrt{K}] - \Phi(x)| \leq C_1 \left( \frac{A}{n^{1/A} T_K} \right)^{1/5} \quad (5.10)$$

where  $C_1$  is an absolute constant.

**Proof:** The proof uses Theorem 5.2, which requires that the martingale be normalized so that the total conditional variance  $V_N^2$  is close to 1. So let  $Y_i = \frac{\kappa_i X_i}{\sqrt{K}}$  and consider the martingale  $\{\sum_{j=1}^i Y_j, \mathcal{F}_i\}$ . To apply the theorem we need to compute a bound on the value  $L_N$ . We will fix  $\delta = 1$ .

We begin by getting a bound on the first term  $\sum \mathbb{E}[|Y_i|^{2+2\delta}]$ . We have

$$\sum_{i=1}^N \mathbb{E}[|Y_i|^4] = \mathbb{E}\left[\sum_{i=1}^N |Y_i|^4\right] = \frac{1}{K^2} \mathbb{E}\left[\sum_{i=1}^N |\kappa_i X_i|^4\right] = \frac{1}{K^2} \mathbb{E}\left[\sum_{i=1}^n \sum_{j=1}^{T_i} j^{4a}\right] \quad (5.11)$$

Now,

$$\sum_{j=1}^{T_i} j^{4a} \leq \int_0^{T_i} j^{4a} dj + T_i^{4a} = \frac{T_i^{4a+1}}{4a+1} + T_i^{4a}.$$

Define  $\psi(x) = x^A/A$ ,  $\chi(x) = x^{4a} + \frac{x^{4a+1}}{4a+1}$ , taking  $0^0 = 1$ . Then  $\chi\psi^{-1}(y) = (Ay)^{4a/A} + \frac{(Ay)^{(4a+1)/A}}{4a+1}$  is convex, and hence  $\sum_{i=1}^n \left(T_i^{4a} + \frac{T_i^{4a+1}}{4a+1}\right)$  is at most  $(n^{1/A} T_K)^{4a} + \frac{(n^{1/A} T_K)^{4a+1}}{4a+1} + (n-1)\chi(0)$  using Lemma 5.8. If  $a$  is positive then  $\chi(0)$  is zero; however if  $a$  is zero  $\chi(0)$  will be 1. In either case  $(n-1)\chi(0) \leq n-1$ . Plugging everything value back into (5.11) gives

$$\sum_{i=1}^N \mathbb{E}[|Y_i|^4] \leq \frac{(n^{1/A} T_K)^{4a}}{K^2} + \frac{(n^{1/A} T_K)^{4a+1}}{K^2(4a+1)} + \frac{n-1}{K^2}. \quad (5.12)$$

For the second term  $\mathbb{E}[|V_N^2 - 1|^{1+\delta}]$ , observe that

$$V_N^2 = \sum_{i=1}^N \mathbb{E}[Y_i^2 | \mathcal{F}_{i-1}] = \frac{1}{K} \sum_{i=1}^N \mathbb{E}[(\kappa_i X_i)^2 | \mathcal{F}_{i-1}],$$

which is just  $1/K$  times the sum of the squares of the weights  $|\kappa_i X_i|$  of the common votes. But the total variance of the common votes can differ from  $K$  by at most the variance of the first vote  $X_i$  for which  $\kappa_i = 0$ . Since the processor that casts this vote can have cast at most  $n^{1/A} T_K$  votes beforehand, the variance of this vote is at most  $(n^{1/A} T_K + 1)^{2a}$ , giving the bound

$$|V_N^2 - 1|^{1+\delta} \leq \frac{1}{K} (n^{1/A} T_K + 1)^{2a}. \quad (5.13)$$

Combining (5.12) and (5.13) gives

$$\begin{aligned}
L_N &\leq \frac{(n^{1/A}T_K)^{4a}}{K^2} + \frac{(n^{1/A}T_K)^{4a+1}}{K^2(4a+1)} + \frac{n-1}{K^2} + \frac{(n^{1/A}T_K+1)^{2a}}{K} \\
&= \frac{n^{4a/A}T_K^{4a}}{K^2} + \frac{n^{(4a+1)/A}T_K^{4a+1}}{K^2(4a+1)} + \frac{A^2(n-1)}{n^2T_K^{2A}} \\
&\quad + \frac{n^{2a/A}T_K^{2a}(1+n^{-1/A}T_K^{-1})^{2a}}{K} \\
&\leq A^2n^{-2/A}T_K^{-2} + \frac{A^2n^{-1/A}T_K^{-1}}{4a+1} + A^2n^{-1}T_K^{-2A} \\
&\quad + An^{-1/A}T_K^{-1}e^{n^{-2a/A}T_K^{-2a}} \\
&\leq \frac{6A^2}{n^{1/A}T_K}
\end{aligned}$$

The second-to-last step uses the approximation  $(1+x)^b \leq e^{bx}$  for non-negative  $b$  and  $x$ . The exponential term is serendipitously bounded by  $e$  if (5.9) holds, since  $6A^2(n^{1/A}T_K)^{-1} \leq 1$  implies that  $(n^{1/A}T_K)^{-2a}$  is also at most 1.

A more direct application of (5.9) shows that  $L_N \leq 1$ , and thus Theorem 5.2 applies. Hence

$$\begin{aligned}
\left| \Pr\left[\sum \kappa_i X_i \leq x\sqrt{K}\right] - \Phi(x) \right| &= \left| \Pr\left[\sum_{i=1}^N Y_i \leq x\right] - \Phi(x) \right| \\
&\leq C \left( \frac{6A^2}{n^{1/A}T_K} \right)^{1/5} \left( \frac{1}{1+|x|^{16/5}} \right) \\
&\leq C_1 \left( \frac{A^2}{n^{1/A}T_K} \right)^{1/5}.
\end{aligned}$$

### 5.4.3 Extra votes

In this section we examine the extra votes from the point of view of a particular processor  $P$ .

Recall that  $\zeta_{P,i}$  is defined to be 1 if the vote  $X_i$  is cast by some processor  $Q$  before  $P$ 's final read of  $Q$ 's register and 0 otherwise. Clearly,  $\zeta_{P,i} \geq \kappa_i$  since  $P$

could not have started its final read until the total variance exceeded  $K$ . As discussed above, both  $\zeta_{P,i}$  and  $\kappa_i$  are  $\mathcal{F}_{i-1}$ -measurable. Thus  $\xi_i = \zeta_{P,i} - \kappa_i$  is a  $0-1$  random variable that is  $\mathcal{F}_{i-1}$ -measurable, and  $\{S_{P,i} = \sum_{j=1}^i \xi_j X_j, \mathcal{F}_i\}$  is a martingale by Theorem 5.1.

Define  $\Delta = n(gT_K)^a$ . The following lemma shows a bound on the tails of  $\sum \xi_i X_i$ .

**Lemma 5.12** *For any  $x > 0$ , if*

$$g^a \leq d \sqrt{\frac{T_K}{nA}} \quad (5.14)$$

*holds for some positive  $d < x$ , and*

$$g^A \leq 1 + \frac{(x-d)^2}{2 \log(n/p)} \quad (5.15)$$

*holds for some positive  $p < n$ , then for each processor  $P$ ,*

$$\Pr\left[\sum (\zeta_{P,i} - \kappa_i) X_i \leq \Delta - x\sqrt{K}\right] \leq p/n. \quad (5.16)$$

**Proof:** The proof uses Corollary 5.5, so we proceed by showing that its premises (stated in Theorem 5.4) are satisfied.

By Corollary 5.10,  $X_i$  and thus  $\xi_i X_i$  is zero for  $i > n(T_K + c + 1)$ . So  $\sum \xi_i X_i = S_{P,M}$  where  $M = n(T_K + c + 1)$ .

Set  $w_i = |\xi_i X_i|$ . Then the first premise of Corollary 5.5 follows from the fact that for each  $i$ ,  $\xi_i$  and  $|X_i|$  are both  $\mathcal{F}_i$ -measurable. The second premise is immediate. For the third premise, notice that

$$\sum (|\xi_i X_i|)^2 = \sum \xi_i X_i^2 = \sum \zeta_{P,i} X_i^2 - \sum \kappa_i X_i^2 \leq \sum X_i^2 - \sum \kappa_i X_i^2.$$

The first term is

$$\sum X_i^2 = \sum_{i=1}^n \sum_{j=1}^{U_i} j^{2a}.$$

The second term is

$$\sum \kappa_i X_i^2 \geq K - t^{2a}$$

for some  $t$  which is at most  $U_i$  for some  $i$ . Thus

$$\begin{aligned}
 \sum(|\xi_i X_i|)^2 &\leq -K + t^{2a} + \sum_{i=1}^n \sum_{j=1}^{U_i} j^{2a} \\
 &< -K + \sum_{i=1}^n \sum_{j=1}^{U_i+1} j^{2a} \\
 &\leq -K + \sum_{i=1}^n (U_i + 2)^A / A.
 \end{aligned} \tag{5.17}$$

Let  $\chi(x) = (x + 2)^A / A$ . Then

$$\begin{aligned}
 \chi(\psi^{-1}(y) + c + 1) &= \frac{((Ay)^{1/A} + c + 3)^A}{A} \\
 &= \frac{1}{A} \sum_{k=0}^A \binom{A}{k} (Ay)^{k/A} (c + 3)^{A-k}
 \end{aligned}$$

For  $y \geq 0$ , the second derivative of each term is either 0 (when  $k = A$ ) or negative; thus  $\chi(\psi^{-1}(y) + c + 1)$  is concave and Lemma 5.9 gives

$$\sum_{i=1}^n \frac{(U_i + 2)^A}{A} \leq n\chi(T_K + c + 1) = \frac{n(T_K + c + 3)^A}{A} \leq \frac{n(gT_K)^A}{A}. \tag{5.18}$$

It follows from (5.17) and (5.18) that

$$\sum(|\xi_i X_i|)^2 \leq \frac{n(gT_K)^A}{A} - K = K(g^A - 1)$$

Applying (5.4) from Corollary 5.5 now yields, for all  $\lambda > 0$ ,

$$\Pr[S_{P,M} \leq -\lambda] \leq e^{-\lambda^2/2K(g^A - 1)}. \tag{5.19}$$

If (5.14) holds, then  $\Delta \leq d\sqrt{K}$  by Lemma 5.13. So

$$\begin{aligned}
 \Pr\left[\sum \xi_i X_i \leq \Delta - x\sqrt{K}\right] &\leq \Pr[S_{P,M} \leq -(x - d)\sqrt{K}] \\
 &\leq e^{-(x-d)^2 K/2K(g^A - 1)} \\
 &= e^{-(x-d)^2/2(g^A - 1)}.
 \end{aligned}$$

But if (5.15) holds then

$$g^A - 1 \leq \frac{(x - d)^2}{2 \log(n/p)}$$

and, since  $\log(n/p) > 0$  and  $g > 1$ ,

$$-\frac{(x - d)^2}{2(g^A - 1)} \leq -\log(n/p) = \log(p/n),$$

From which it follows that

$$e^{-(x-d)^2/2(g^A-1)} \leq e^{\log(p/n)} = p/n.$$

■

#### 5.4.4 Written votes vs. decided votes

In this section we show that the difference between  $\sum \zeta_{P,i} X_i$  and the total vote actually read by  $P$  is bounded by  $\Delta = n(gT_K)^a$ .

**Lemma 5.13** *Let  $R_P$  be the sum of the votes read during  $P$ 's final read. Then*

$$\left| \sum \zeta_{P,i} X_i - R_P \right| \leq n(T_k + c + 1)^a \leq n(gT_K)^a = \Delta \quad (5.20)$$

**Proof:** Suppose  $\zeta_{P,i} = 1$ , and suppose  $X_i$  is decided by processor  $P_j$ . If the vote  $X_i$  is not included in the value read by  $P$ , it must have been decided before  $P$ 's read of  $P_j$ 's register but written afterwards. Because each vote is written out before the next vote is decided there can be at most one vote from  $P_j$  which is included in  $\sum \zeta_{P,i} X_i$  but is not actually read by  $P$ . This vote has weight at most  $U_j^a$ . So we have  $|\sum \zeta_{P,i} X_i - R_P| \leq \sum_{i=1}^n U_i^a$ . Now let  $\chi(x) = x^a$ . Then

$$\chi(\psi^{-1}(y) + c + 1) = ((Ay)^{1/A} + c + 1)^a = \sum_{k=0}^a \binom{a}{k} (Ay)^{k/A} (c + 1)^{a-k}$$

which is concave since the second derivative of each term of the sum is negative. The rest follows from Lemma 5.9. ■

### 5.4.5 Choice of parameters

Let us summarize the proof of correctness in a single theorem:

**Theorem 5.14** *Define*

$$\begin{aligned} A &= 2a + 1 \\ T_K &= \left(\frac{AK}{n}\right)^{1/A} \\ g &= 1 + \frac{c+3}{T_K} \end{aligned}$$

*and suppose there exist  $d > 0$ ,  $x > d$  and positive  $p < n$  such that all of the following hold:*

$$g^a \leq d\sqrt{\frac{T_K}{nA}} \quad (5.21)$$

$$g^A \leq 1 + \frac{(x-d)^2}{2\log(n/p)} \quad (5.22)$$

$$\frac{6A^2}{n^{1/A}T_K} \leq 1 \quad (5.23)$$

*Then the protocol implements a shared coin with agreement parameter at least*

$$1 - \left[ \Phi(x) + C_1 \left( \frac{A^2}{n^{1/A}T_K} \right)^{1/5} + p \right] \quad (5.24)$$

*where  $C_1$  is the constant from Lemma 5.11.*

**Proof:** To show that the agreement parameter is at least (5.24) we must show that for each  $z \in \{0, 1\}$  the probability that all processors decide  $z$  is at least (5.24). Without loss of generality let us consider only the probability that all processors decide 1; the case of all processors deciding 0 follows by symmetry.

Recall the definition  $\Delta = n(gT_K)^a$ . Suppose that  $\sum \kappa_i X_i > x\sqrt{K}$ , and that for each processor  $P$ ,  $\sum (\zeta_{P,i} - \kappa_i) X_i > \Delta - x\sqrt{K}$ . Then for each  $P$  we have  $\sum \zeta_{P,i} X_i > \Delta$  and by Lemma 5.13  $P$  reads a value greater than 0 during its final read and thus decides 1.

Now for this event *not* to occur, we must either have  $\sum \kappa_i X_i \leq x\sqrt{K}$  or  $\sum (\zeta_{P,i} - \kappa_i) X_i \leq \Delta - x\sqrt{K}$  for some  $P$ . But as the probability of a union of events never exceeds the sum of the probabilities of the events, the probability of failing in any of these ways is at most

$$\begin{aligned} & \Pr\left[\sum \kappa_i X_i \leq x\sqrt{K}\right] + \sum_P \Pr\left[\sum (\zeta_{P,i} - \kappa_i) X_i \leq \Delta - x\sqrt{K}\right] \\ & \leq \left[\Phi(x) + C_1 \left(\frac{A^2}{n^{1/4} T_K}\right)^{1/5}\right] + n(p/n) \end{aligned} \quad (5.25)$$

by Lemmas 5.11 and 5.12. So the probability some processor decides 0 is at most (5.25), and thus the probability that all processors decide 1 is at least 1 minus (5.25). ■

The running time of the protocol is more easily shown:

**Theorem 5.15** *No processor executes more than  $(AK)^{1/4}(2+n/c)+2c+2n$  register operations during an execution of the shared coin protocol.*

**Proof:** First consider the maximum number of votes a processor can cast. After  $(AK)^{1/4}$  votes the total variance of the processor's votes will be

$$\sum_{x=1}^{(AK)^{1/4}} x^{2a} > \int_0^{(AK)^{1/4}} x^{2a} dx = \frac{((AK)^{1/4})^A}{A} = K,$$

so after at most  $c$  votes the processor will execute line 11 of Figure 5.1 and see a total variance greater than  $K$ . Thus each processor casts at most  $(AK)^{1/4} + c$  votes. But each vote costs 1 write operation in line 8, and every  $c$  votes costs  $n$  reads in line 11, to which must be added a one-time cost of  $n$  reads in line 13. The total number of operations is thus at most  $((AK)^{1/4} + c)(1 + \lceil n/c \rceil) + n \leq ((AK)^{1/4} + c)(2 + n/c) + n = (AK)^{1/4}(2 + n/c) + 2c + 2n$ . ■

It remains only to find values for  $a$ ,  $K$ , and  $c$  which give both a constant agreement parameter and a reasonable running time. As a warm-up, let us consider what happens if we emulate the protocol of Bracha and Rachman [BR91]:

**Theorem 5.16** *If  $a = 0$ ,  $K = 4n^2$ , and  $c = \frac{n}{4\log n} - 3$ , then for  $n$  sufficiently large the protocol implements a shared coin with agreement parameter at least 0.05 in which each processor executes at most  $O(n^2 \log n)$  operations.*

**Proof:** For the agreement parameter, we have  $A = 1$ ,  $T_K = 4n$ , and  $g = 1 + 1/16 \log n$ . Let  $d = 1/2$ ,  $x = 1$ , and  $p = 1/10$ . Then (5.21) holds since  $g^a = 1 \leq d\sqrt{T_K/nA} = 1$ . Furthermore,

$$\begin{aligned} \left(1 + \frac{(x-d)^2}{2 \log(n/p)}\right)^{1/A} &= 1 + \frac{1}{8(\log n - \log p)} \\ &\geq 1 + \frac{1}{16 \log n} \end{aligned}$$

when  $n^2 > 1/p$ . Thus (5.22) holds. The remaining inequality (5.23) holds for  $n \geq 2$ , so by Theorem 5.14 we have a probability of failure of at most

$$\begin{aligned} \Phi(1) + C_1 \left(\frac{1}{4n^2}\right)^{1/5} + p \\ \leq 0.842 + O\left(\frac{1}{n^{2/5}}\right) + 0.1 \end{aligned}$$

which is not more than  $0.942 + \epsilon$  for  $n$  sufficiently large. In particular for  $n$  greater than some  $n_0$  this quantity is at most 0.95, and the agreement parameter is thus at least  $1 - 0.95$ .

The running time is immediate from Theorem 5.15. ■

Now consider what happens if  $a$  is not restricted to be a constant 0.

**Theorem 5.17** *If  $a = (\log n - 1)/2$ ,  $K = (16n \log n)^{\log n} (n/\log n)$ , and  $c = n/\log n - 3$ , then for  $n$  sufficiently large the protocol implements a shared coin with constant agreement parameter in which each processor executes at most  $O(n \log^2 n)$  operations.*

**Proof:** We have  $A = \log n$ ,  $T_K = 16n \log n$ , and  $g = 1 + \frac{1}{16 \log^2 n}$ . Let  $d = 1/2$ ,  $x = 1$ , and  $p = 1/10$ .

We want to apply Theorem 5.14, so first we verify that its premises are satisfied. To show (5.21), compute

$$g^a = \left(1 + \frac{1}{16 \log^2 n}\right)^{(\log n - 1)/2} \leq e^{(\log n - 1)/32 \log^2 n} \leq e^{1/32 \log n}$$

which for  $n \geq 2$  will be less than  $d\sqrt{T_K/nA} = 2$ . To show (5.22), note that

$$g^A = \left(1 + \frac{1}{16 \log^2 n}\right)^{\log n} \leq e^{1/16 \log n}$$

and thus  $\log(g^A) \leq 1/16 \log n$ . But

$$\begin{aligned} \log \left(1 + \frac{(x-d)^2}{2 \log(n/p)}\right) &= \log \left(1 + \frac{1}{8 \log(n/p)}\right) \\ &\geq \frac{1}{8 \log(n/p)} - \frac{1}{128 \log^2(n/p)} \\ &= \frac{1}{8(\log n - \log p)} - \frac{1}{128(\log n - \log p)^2} \end{aligned}$$

(using the approximation  $\log(1+x) \geq x - \frac{1}{2}x^2$ ). For sufficiently large  $n$  this quantity exceeds  $1/16 \log n$  and (5.22) holds. The remaining constraint (5.23) is easily verified, and thus Theorem 5.14 applies and the agreement parameter is at least

$$\begin{aligned} 1 - \left[ \Phi(1) + C_1 \left( \frac{\log^2 n}{n^{1/\log n} (16n \log n)} \right)^{1/\varepsilon} + 1/10 \right] \\ \leq 1 - (0.842 + O((\log n/n)^{1/5}) + 0.10) \end{aligned}$$

which is at least 0.05 for sufficiently large  $n$ . Thus the protocol gives a constant agreement parameter.

Now by Theorem 5.15, the number of operations executed by any single processor is at most  $(AK)^{1/A}(2 + n/c) + 2c + 2n$ , or

$$(\log n)^{1/\log n} (16n \log n) (n/\log n)^{1/\log n} O(\log n) + O(n)$$

which is  $O(n \log^2 n)$ . ■

It follows immediately that plugging a coin with the parameters of Theorem 5.17 into the consensus protocol construction of Chapter 3 gives a consensus protocol that requires an expected  $O(n \log^2 n)$  operations per processor. It is not difficult to see that the best bound we can place on the total number of operations is in fact  $n$  times this quantity, or  $O(n^2 \log^2 n)$ . The worst case is when each processor casts the same number of common votes.

# Chapter 6

## Conclusions and Open Problems

In this thesis I have shown:

- A simple algorithm for a robust wait-free shared coin with bias at most  $\epsilon$  which runs in an expected  $O(n^4/\epsilon^2)$  total register operations.
- A modification of this algorithm that achieves consensus in an expected  $O(n^4)$  total register operations, and which can be implemented using only three atomic counters.
- The asymptotically fastest known wait-free consensus protocol in the per-processor measure, based on a shared coin that requires only an expected  $O(n \log^2 n)$  register operations per processor to achieve a constant agreement parameter.

This chapter discusses how these results fit into the history of wait-free consensus, and what difficulties need to be overcome to make further improvements. It concludes with a list of open problems.

### 6.1 Comparison with other protocols

Table 6.1 gives a comparison of the running times of wait-free consensus protocols for the shared-memory model. In this table the quantity  $p$  is the

	Expected operations	
	Per processor	Total
Abrahamson [Abr88]	$2^{O(n^2)}$	$2^{O(n^2)}$
Aspnes and Herlihy [AH90a]	$O(n^4)$	$O(n^4)$
Attiya, Dolev, and Shavit [ADS89]	$O(n^4)$	$O(n^4)$
<b>Chapter 4 ([Asp90])</b>	$O(n^2(p^2 + n))$	$O(n^2(p^2 + n))$
Bracha and Rachman [BR90]	$O(n(p^2 + n))$	$O(n(p^2 + n))$
Dwork et al. [DHPW92]	$O(n(p^2 + n))$	$O(n(p^2 + n))$
Saks, Shavit, and Woll [SSW91]	$O(n^4)$	$O(n^4)$
Bracha and Rachman [BR91]	$O(n^2 \log n)$	$O(n^2 \log n)$
<b>Chapter 5 ([AW92])</b>	$O(n \log^2 n)$	$O(n^2 \log^2 n)$

Table 6.1: Comparison of consensus protocols.

number of *active* processors as defined in Section 4.4. The first known protocol was the exponential protocol of Abrahamson [Abr88]. The first known polynomial-time protocol was that of Aspnes and Herlihy [AH90a]. Attiya, Dolev, and Shavit [ADS89] described a modification of this protocol which required only a bounded amount of space, but which retained the spirit of the rounds-based structure of the Aspnes-Herlihy protocol.

The protocol of Chapter 4, which also appears in [Asp90], was the first to eliminate the use of rounds by using a robust shared coin. Since its first appearance its performance was improved by a factor of  $n$  by Bracha and Rachman [BR90] and by Dwork et al. [DHPW92]. Both groups achieved the improvement by replacing the  $O(n^2)$  implementation of an atomic counter with a weaker primitive that required only  $O(n)$  register operations per counter operation, and acted sufficiently like a counter to make the consensus protocol work.

The first protocol to use the idea of casting votes until a quorum is reached (instead of until a sufficiently large margin of victory is reached) was that of Saks, Shavit, and Woll [SSW91]. Their protocol was optimized for the special case where nearly all of the processors are running in lockstep. Bracha and Rachman [BR91] noticed that the protocol could be sped up by having each processor read all the registers only after every  $O(n/\log n)$  votes; the resulting protocol is a special case of the protocol of Chapter 5 obtained by setting  $a$  to 0. The protocol of Chapter 5, which also appears in [AW92], is the first to use votes of unequal weight, and as a result is the first for which

the maximum expected number of operations executed by a single processor is more than a constant factor less than the maximum executed by all of the processors together.

## 6.2 Limits to wait-free consensus

The table shows a considerable evolution of wait-free consensus protocols since Abrahamson's exponential solution. It is natural to ask how much better consensus protocols can still get.

One limitation we quickly run into is the following. If a processor is running by itself, it must read every other processor's register at least once. If not, it cannot distinguish the situation where it really ran first all by itself from the situation where some other processor (whose register it has not read) ran to completion before it started. In the latter case the processor would be required by the consistency condition to agree with its unseen predecessor: but without reading that predecessor's register it would have no way of knowing which value to choose. Thus in any wait-free consensus protocol some processor can always be forced to execute at least  $n - 1$  read operations. This  $\Omega(n)$  lower bound is unaffected even if the adversary is substantially weakened; the argument remains valid, for example, if the adversary is not allowed to see the internal states of processors or even if it is required to specify all of its scheduling decisions before the protocol starts. So in fact the  $O(n \log^2 n)$  protocol we have described here is close to the best we can hope for in the per-processor measure, given the assumption of single-writer registers, even against relatively weak adversaries.

On the other hand, the question of how far the total number of operations can be reduced does not have as easy an answer. That some processor can be forced to execute  $\Omega(n)$  operations does not mean that all processors can be forced to; it could be the case that if the processors cooperate they could collectively gather information about the state of the system faster than they would independently. In fact, the best known lower bound for expected total operations is only  $\Omega(n \log n)$ , based on the minimum number of operations needed to communicate every processor's state to every other processor [SSW91]. Furthermore, the fact that the protocol of [BR91] achieves a bound of  $O(n^2 \log n)$  on total work shows that some improvement is possible on our protocol, though possibly only at the expense of increasing

the per-processor bound.

However, to get below  $\Omega(n^2)$  operations will require at least two breakthroughs. The first problem is that all of the algorithms we currently have require that every processor read every other processor's register directly at some point, which takes  $\Theta(n^2)$  total operations. It seems likely that some sort of randomized cooperative technique could allow this dissemination of information to proceed more quickly (possibly at the cost of using very large registers); but at present no such technique is known.

The second problem is that to reduce the total number of operations below  $\Omega(n^2)$  it will be necessary to reduce the number of local random choices below  $\Omega(n^2)$ , as local coin-flips that have no writes between them effectively consolidate into a single random choice from the point of view of the scheduler. This problem appears more difficult than the first, as it requires abandoning the voting technique at the heart of all currently known wait-free consensus protocols. The reason is that in these protocols, the scheduler's power only becomes limited when the standard deviation of the total vote becomes comparable to the sum of the votes that the scheduler can withhold. With unweighted votes,  $\Omega(n^2)$  votes are required; for weighted votes the situation is only made worse, as increasing the weight of some votes increases the sum of the withheld votes more quickly than it increases the standard deviation of the total vote. It appears that it will be difficult to get below  $\Omega(n^2)$  without adopting some decision method that takes more account of the ordering of events in the system.

### 6.3 Open problems

The consensus protocol described in Chapter 5 comes quite close to the limits of current methods for solving wait-free consensus. Aside from optimizations such as eliminating the  $\log n$  factors from the per-processor bound or reducing the value of  $n$  at which the protocol becomes practical, essentially the only question remaining is whether the total number of operations can be reduced substantially. There are several questions whose answers would shed light on this problem, as well as many other problems in the area:

1. Is it possible in the asynchronous shared-memory model for  $n$  processors to collectively read  $n$  registers in fewer than  $\Theta(n^2)$  total operations?

2. Does every consensus protocol contain a shared coin?
3. Can a shared coin with constant agreement parameter be built that requires less than  $\Omega(n^2)$  total operations? (A closely related question: can a shared coin of arbitrarily small bias  $\epsilon$  run in less than  $\Omega(n^2/\epsilon^2)$  total operations?)

# Bibliography

[AAD<sup>+</sup>90] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. In *Proceedings of the Ninth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 1–14, August 1990.

[Abr88] Karl Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 291–302, August 1988.

[ADS89] H. Attiya, D. Dolev, and N. Shavit. Bounded polynomial randomized consensus. In *Proceedings of the Eighth ACM Symposium on Principles of Distributed Computing*, pages 281–294, August 1989.

[AFL83] Eshrat Arjomandi, Michael J. Fischer, and Nancy A. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *Journal of the ACM*, 30(3):449–456, July 1983.

[AG91] James H. Anderson and Bojan Grošelj. Beyond atomic registers: Bounded wait-free implementations of non-trivial objects. In *Proceedings of the Fifth International Workshop on Distributed Algorithms*, pages 52–70. Springer-Verlag, 1991.

[AH90a] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, September 1990.

[AH90b] James Aspnes and Maurice Herlihy. Wait-free synchronization in the asynchronous PRAM model. In *Second Annual ACM Symposium on Parallel Algorithms and Architectures*, July 1990.

[ALS90] Hagit Attiya, Nancy Lynch, and Nir Shavit. Are wait-free algorithms fast? In *31st Annual Symposium on Foundations of Computer Science*, pages 55–64, October 1990.

[And90] James H. Anderson. Composite registers. In *Proceedings of the Ninth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 15–29, August 1990.

[AS92] Noga Alon and Joel H. Spencer. *The Probabilistic Method*. John Wiley and Sons, 1992.

[Asp90] James Aspnes. Time- and space-efficient randomized consensus. In *Proceedings of the Ninth ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 325–331, August 1990. To appear, *Journal of Algorithms*.

[AW92] James Aspnes and Orli Waarts. Randomized consensus in expected  $O(n \log^2 n)$  operations per processor. To appear, *Thirty-Third Annual Symposium on Foundations of Computer Science*, 1992.

[BD88] Shai Ben-David. The global time assumption and semantics for concurrent systems. In *Proceedings of the Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 223–231, August 1988.

[Bil86] Patrick Billingsley. *Probability and Measure*. John Wiley and Sons, second edition, 1986.

[BND89] Amotz Bar-Noy and Danny Dolev. Shared-memory vs. message-passing in an asynchronous distributed environment. In *Proceedings of the Eighth ACM Symposium on Principles of Distributed Computing*, pages 307–318, August 1989.

[BP87] James E. Burns and Gary L. Peterson. Constructing multi-reader atomic values from non-atomic values. In *Proceedings of the Sixth*

*ACM Symposium on Principles of Distributed Computing*, pages 222–231, 1987.

- [BR90] Gabi Bracha and Ophir Rachman. Approximated counters and randomized consensus. Technical Report 662, Technion, 1990.
- [BR91] Gabi Bracha and Ophir Rachman. Randomized consensus in expected  $O(n^2 \log n)$  operations. In *Proceedings of the Fifth International Workshop on Distributed Algorithms*. Springer-Verlag, 1991.
- [BT83] Gabriel Bracha and Sam Toueg. Asynchronous consensus and byzantine protocols in faulty environments. Technical Report 83-559, Department of Computer Science, Cornell University, 1983.
- [Che52] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23(4):493–407, December 1952.
- [CIL87] Benny Chor, Amos Israeli, and Ming Li. On processor coordination using asynchronous hardware. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 86–97, 1987.
- [CM89] Benny Chor and Lior Moscovici. Solvability in asynchronous environments. In *30th Annual Symposium on Foundations of Computer Science*, pages 422–427, October 1989.
- [CMS89] Benny Chor, Michael Merritt, and David B. Shmoys. Simple constant-time consensus protocols in realistic failure models. *Journal of the ACM*, 36(3):591–614, July 1989.
- [DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.
- [DHPW92] Cynthia Dwork, Maurice Herlihy, Serge Plotkin, and Orli Waarts. Time-lapse snapshots. In *Proceedings of Israel Symposium on the Theory of Computing and Systems*, 1992.

[DS89] Danny Dolev and Nir Shavit. Bounded concurrent timestamp systems are constructible! In *Proceedings of the Twenty-First Annual ACM Symposium on the Theory of Computing*, pages 454–465, 1989.

[DW92] Cynthia Dwork and Orli Waarts. Simple and efficient bounded concurrent timestamping or bounded concurrent timestamp systems are comprehensible! In *Proceedings of the Twenty-Fourth Annual ACM Symposium on the Theory of Computing*, pages 655–666, 1992.

[DY75] E.B. Dynkin and A.A. Yushkevich. *Controlled Markov Processes*. Springer-Verlag, 1975.

[Fel68] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. John Wiley and Sons, third edition. 1968.

[Fel71] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 2. John Wiley and Sons, second edition. 1971.

[FLP85] Michael J. Fischer, Nancy Ann Lynch, and Michael S. Paterson. Impossibility of distributed commit with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[Hal39] Paul R. Halmos. Invariants of certain stochastic transformations: The mathematical theory of gambling systems. *Duke Mathematical Journal*, 5(2):461–478, June 1939.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[HH80] P. Hall and C.C. Heyde. *Martingale Limit Theory and Its Application*. Academic Press, 1980.

[HLP52] G. Hardy, J.E. Littlewood, and G. Pólya. *Inequalities*. Cambridge University Press, second edition, 1952.

[IL87] Amos Israeli and Ming Li. Bounded time stamps. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 371–382, 1987.

[Kop84] P.E. Kopp. *Martingales and Stochastic Integrals*. Cambridge University Press, 1984.

[LAA87] Michael C. Loui and Hosame H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. In Franco P. Preparata, editor, *Advances in Computing Research*, volume 4. JAI Press, 1987.

[Lam77] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[Lam86a] Leslie Lamport. On interprocess communication, part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[Lam86b] Leslie Lamport. On interprocess communication, part II: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[LF81] Nancy A. Lynch and Michael J. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13:17–43, 1981.

[LL90] Leslie Lamport and Nancy Lynch. Distributed computing: Models and methods. In Jan Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter IX, pages 1157–1199. MIT Press, 1990.

[Lyn88] Nancy Lynch. I/O automata: A model for discrete event systems. Technical Report MIT/LCS/TM-351, MIT Laboratory for Computer Science, March 1988.

[NW87] Richard Newman-Wolfe. A protocol for wait-free, atomic, multi-reader shared variables. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 232–249, 1987.

[Pet83] Gary L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[Plo89] Serge A. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the Eighth ACM Symposium on Principles of Distributed Computing*, pages 159–176, August 1989.

[SAG87] Ambuj K. Singh, James H. Anderson, and Mohamed G. Gouda. The elusive atomic register revisited. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 206–221, August 1987.

[Spe87] Joel Spencer. *Ten Lectures on the Probabilistic Method*. Society for Industrial and Applied Mathematics, 1987.

[SSW91] Michael Saks, Nir Shavit, and Heather Woll. Optimal time randomized consensus – making resilient algorithms fast in practice. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 351–362, 1991.

[SV86] Miklos Santha and Umesh V. Vazirani. Generating quasi-random sequences from semi-random sources. *Journal of Computer and System Sciences*, 33:75–87, 1986.

[TM89] Gadi Taubenfeld and Shlomo Moran. Possibility and impossibility results in a shared memory environment. In *Proceedings of the Third International Workshop on Distributed Algorithms*, pages 254–267. Springer-Verlag, September 1989.